VAX/VMS I/O User's Reference Manual: Part II

Order No. AA-Z601B-TE

digital software

VAX/VMS I/O User's Reference Manual: Part II

Order Number: AA–Z601B–TE

April 1986

This document contains the information necessary to interface directly with the communications I/O device drivers supplied as part of the VAX/VMS operating system. Several examples of programming techniques are included. This document does not contain information on I/O operations using VAX Record Management Services.

Revision/Update Information:	This document supersedes the *VAX/VMS I/O User's Reference Manual: Part II* Version 4.0.
Operating System and Version:	VAX/VMS Version 4.4

digital equipment corporation
maynard, massachusetts

April 1986

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital

ZK–3066

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

Contents

FIGURES

TABLES

Contents

Preface

Manual Objectives

This manual provides users of the VAX/VMS operating system with the information they need to interface directly with I/O device drivers supplied as part of the operating system. It is not the objective of this manual to provide the reader with information on all aspects of VAX/VMS input/output (I/O) operations.

Intended Audience

This manual is intended for system programmers who wish to take advantage of the time and space savings that result from direct use of the I/O devices. Users of the VAX/VMS operating system who do not require such detailed knowledge of the I/O drivers can use the device-independent services described in the *VAX Record Management Services Reference Manual*. Readers are expected to have some experience with either VAX MACRO or some other high-level assembly language.

Structure of This Document

This manual is organized into six sections and one appendix, as follows:

- Sections 1 through 6 describe the use of communications device drivers supported by VAX/VMS.
  - Section 1 discusses the DMC11/DMR11 interface driver.
  - Section 2 discusses the DMP11, DMF32, and asynchronous DDCMP interface drivers.
  - Section 3 discusses the DR11–W and DRV11-WA interface driver.
  - Section 4 discusses the DR32 interface driver.
  - Section 5 discusses the DUP11 interface driver.
  - Section 6 discusses the DEUNA, DEQNA, and DELUA device drivers.
- The appendix summarizes the function codes, arguments, and function modifiers used by the drivers listed previously.

Associated Documents

The following documents may also be useful.

- *General Information Volume*—contains a complete list of all VAX/VMS documents and a master index of all topics discussed in the VAX/VMS document set
- *VAX/VMS System Services Reference Manual*

- *VAX Software Handbook*
- *PDP-11 Peripherals Handbook*
- *VAX FORTRAN User's Guide*
- *Guide to Programming on VAX/VMS*
- *VAX Record Management Services Reference Manual*
- *VAX/VMS Networking Manual*
- *VAX-11 2780/3780 Protocol Emulator User's Guide*
- *VAX/VMS System Messages and Recovery Procedures Reference Manual*

Conventions Used in This Document

Convention	Meaning
[ ]	Brackets in QIO requests enclose optional arguments. For example: `IO$_SETCHAR P1,[P2],P3,[P6]`
. . .	Horizontal ellipsis indicate that characters or arguments not pertinent to the example have been omitted. For example: This file defines many (but not all) of the XF$ . . . symbolic names described in this section.

.
.
.
	Vertical ellipsis in coding examples indicate that lines of code not pertinent to the example are omitted. For example:
	`LOGNAM: .ASCID /SYS$INPUT/`
	`.`
	`.`
	`.`
	`; DETERMINE TERMINAL NAME`
	`$GETDVI_S -`
	`DEVNAME=LOGNAM, -`
	`ITMLST=DVILIST`

-	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows. For example:
	`CMDOFAB: $FAB fac=put,fnm=sys$output:,-`
	`mrs=132,rat=cr,rfm=var`
	`CMDORAB: $RAB ubf=cmdbuf,usz=cmdbsz,-`
	`fab=cmdofab`

numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated in the coding examples.

Summary of Technical Changes

This revision of the *VAX/VMS I/O User's Reference Manual: Part II* reflects the main technical changes since VAX/VMS Version 4.0. The following sections contain new or changed information:

- Section 1—This section on the DMC11/DMR11 driver now includes information on the Get Device/Volume Information ($GETDVI) system service, which replaces the Get I/O Channel Information ($GETCHN) and Get I/O Device Information ($GETDEV) system services.
- Section 2—This section on the DMP11/DMF32/Asynchronous DDCMP driver now includes information on the $GETDVI system service, which replaces the $GETCHN and $GETDEV system services.
- Section 3—This section on the DR11–W driver now includes information on the DRV11-WA driver. The $GETDVI system service replaces the $GETCHN and $GETDEV system services.
- Section 4—This section on the DR32 driver now includes information on the $GETDVI system service, which replaces the $GETCHN and $GETDEV system services.
- Section 5—This section on the DUP11 driver now includes information on the $GETDVI system service, which replaces the $GETCHN and $GETDEV system services.
- Section 6—This section on the DEUNA and DEQNA drivers now includes information on the DELUA device, which uses the XEDRIVER. The $GETDVI system service replaces the $GETCHN and $GETDEV system services.

1 DMC11/DMR11 Synchronous Communications Line Interface Driver

This section describes the use of the VAX/VMS DMC11 synchronous communications line interface driver. (The DMR11 synchronous communications line interface uses the same driver in DMC compatibility mode; references to the DMC11 driver also imply the use of the DMR11 driver operating in DMC11 compatibility mode.) The DMC11 provides a direct-memory-access (DMA) interface between two computer systems using the DIGITAL Data Communications Message Protocol (see Section 1.1.1). The DMC11 supports DMA data transfers of up to 16K bytes at rates of up to 1 million baud for local operation (over coaxial cable) and 56,000 baud for remote operation (using modems). Both full- and half-duplex modes are supported.

The DMC11 is a message-oriented communications line interface that is used primarily to link two separate but cooperating computer systems.

1.1 Supported DMC11 Synchronous Line Interfaces

Table 1–1 lists the DMC11 options supported by the VAX/VMS operating system.

Table 1–1 Supported DMC11 Options

Type	Use
DMC11-AR with DMC11-FA	remote DMC11 and EIA or V35/DDS
DMC11-AR with DMC11-DA	line unit
DMC11-AL with DMC11-MD	local DMC11 and 1M bps or 56
DMC11-AL with DMC11-MA	bps

1.1.1 DIGITAL Data Communications Message Protocol (DDCMP)

To ensure reliable data transmission, the DIGITAL Data Communications Message Protocol (DDCMP) has been implemented, using a high-speed microprocessor. For remote operations, a DMC11 can communicate with a different type of synchronous interface (or even a different type of computer), provided the remote system has implemented DDCMP.

DDCMP detects errors on the communication line interconnecting the systems using a 16-bit cyclic redundancy check (CRC). Errors are corrected, when necessary, by automatic message retransmission. Sequence numbers in message headers ensure that messages are delivered in the proper order with no omissions or duplications.

The DDCMP specification (Order No. AA-K175A-TC) provides more detailed information on DDCMP.

1.2 Driver Features and Capabilities

DMC11 driver capabilities include the following:

- A nonprivileged QIO interface to the DMC11. (This allows use of the DMC11 as a raw-data channel.)
- Unit attention conditions transmitted through attention ASTs and mailbox messages
- Both full- and half-duplex operation
- Interface design common to all communications devices supported by the VAX/VMS operating system
- Error logging of all DMC11 microprocessor and line unit errors
- Online diagnostics
- Separate transmit and receive quotas
- Assignment of several read buffers to the device

The following sections describe mailbox usage and I/O quotas.

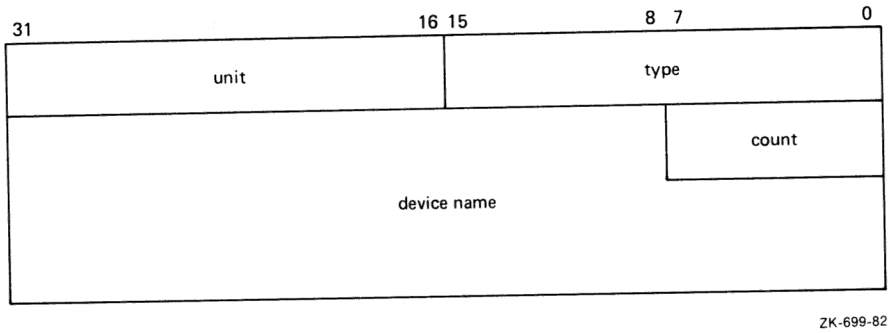
1.2.1 Mailbox Usage

The device owner process can associate a mailbox with a DMC11 by using the Assign I/O Channel ($ASSIGN) system service. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume*.) The mailbox is used to receive messages that signal attention conditions about the unit. As illustrated in Figure 1–1, these messages have the following content and format:

- Message type. This can be any one of the following:
  - MSG$_XM_DATAVL—Data is available.
  - MSG$_XM_SHUTDN—The unit has been shut down.
  - MSG$_XM_ATTN—A disconnect, timeout, or data check occurred.

  The $MSGDEF macro is used to define message types.
- Physical unit number of the DMC11
- Size (count) of the ASCII device name string
- Device name string

Figure 1–1 Mailbox Message Format



1.2.2 Quotas

Transmit operations are considered direct I/O operations and are limited by the process's direct I/O quota.

The quotas for the receive buffer free list (see Section 1.4.3.4) are the process's buffered I/O count and buffered I/O byte limit. After startup, the transient byte count and the buffered I/O byte limit are adjusted.

1.2.3 Power Failure

When a system power failure occurs, no DMC11 recovery is possible. The device is in a fatal error state and is shut down.

1.3 Device Information

Users can obtain information on DMC11/DMR11 device characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume*.)

$GETDVI returns DMC11/DMR11 device characteristics when you specify the item code DVI$_DEVCHAR. Table 1–2 lists these characteristics, which are defined by the $DEVDEF macro.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and class names, which are defined by the $DCDEF macro. The device type for the DMC11 is DT$_DMC11; the device type for the DMR11 is DT$_ DMR11 (only after the device has been started once). The device class for the DMC11 is DC$_SCOM.

DVI$_DEVBUFSIZ returns the maximum message size. The maximum message size is the maximum send or receive message size for the unit. Messages greater than 512 bytes on modem-controlled lines are more prone to transmission errors and therefore may require more retransmissions.

Table 1–2 DMC11/DMR11 Device Characteristics

Characteristic[1]	Meaning
Dynamic Bit (Conditionally Set)	
DEV$M_NET	Network device
Static Bits (Always Set)	
DEV$M_ODV	Output device
DEV$M_IDV	Input device

[1]Defined by the $DEVDEF macro

DVI$_DEVDEPEND returns the DMC11/DMR11 unit characteristics bits, the unit and line status bits, and the error summary bits in a longword field. (Byte 0 = unit characteristics, byte 1 = status, byte 2 = error summary; byte 3 is not used.)

The unit characteristics bits govern the DDCMP operating mode. They are defined by the $XMDEF macro and can be read or set. Table 1–3 lists the unit characteristics values and their meanings.

Table 1–3 DMC11/DMR11 Unit Characteristics

Characteristic	Meaning[1]
XM$M_CHR_MOP	DDCMP maintenance mode
XM$M_CHR_SLAVE	DDCMP half-duplex slave station mode
XM$M_CHR_HDPLX	DDCMP half-duplex mode
XM$M_CHR_LOOPB	DDCMP loopback mode
XM$M_CHR_MBX	The status of the mailbox associated with the unit. If this bit is set, the mailbox is enabled to receive messages signaling unsolicited data. (This bit can also be changed as a subfunction of read or write functions.)

[1]Section 1.1.1 describes DDCMP

The status bits show the status of the unit and the line. The values are defined by the $XMDEF macro. They can be read, set, or cleared as indicated. Table 1–4 lists the status values and their meanings.

Table 1–4 DMC11/DMR11 Unit and Line Status

Status	Meaning
XM$M_STS_ACTIVE	Protocol is active. This bit is set when IO$_SETMODE!IO$_STARTUP is complete, and is cleared when the unit is shut down (read only).
XM$M_STS_TIMO	Timeout. If set, indicates that the receiving computer is unresponsive (read or clear).
XM$M_STS_ORUN	Data overrun. If set, indicates that a message was received but lost because there is no receive buffer (read or clear).

Table 1–4 (Cont.) DMC11/DMR11 Unit and Line Status

Status	Meaning
XM$M_STS_DCHK	Data check. If set, indicates that a retransmission threshold has been exceeded (read or clear).
XM$M_STS_DISC	Disconnect. If set, indicates that the data set ready (DSR) modem line went from on to off (read or clear).

The error summary bits are set only when the driver must shut down the DMC11 interface because a fatal error occurred. These are read-only bits that are cleared by any of the IO$_SETMODE functions (see Section 1.4.3). The XM$M_STS_ACTIVE status bit is clear if any error summary bit is set. Table 1–5 lists the error summary bit values and their meanings.

Table 1–5 DMC11/DMR11 Error Summary Bits

Error Summary Bit	Meaning
XM$M_ERR_MAINT	DDCMP maintenance message was received.
XM$M_ERR_START	DDCMP START message was received.
XM$M_ERR_LOST	Data was lost when a message was received that was longer than the specified maximum message size.
XM$M_ERR_FATAL	An unexpected hardware or software error occurred.

1.4 DMC11 Function Codes

The basic DMC11 function codes are read, write, and set mode. All three functions take function modifiers.

1.4.1 Read

The VAX/VMS operating system provides three read function codes:

- IO$_READLBLK—read logical block
- IO$_READPBLK—read physical block
- IO$_READVBLK—read virtual block

Received messages are multibuffered in system dynamic memory and then copied to the user's address space when the read operation is performed.

The read functions take the following two device/function-dependent arguments:

- P1—the starting virtual address of the buffer that is to receive data
- P2—the size of the receive buffer in bytes

The read functions can take two function modifiers:

- IO$M_DSABLMBX—disables use of the associated mailbox for unsolicited data notification

- IO$M_NOW—completes the read operation immediately if no message is available

1.4.2 Write

The VAX/VMS operating system provides three write function codes:

- IO$_WRITELBLK—write logical block
- IO$_WRITEPBLK—write physical block
- IO$_WRITEVBLK—write virtual block

Transmitted messages are sent directly from the requesting process's buffer.

The write functions take the following two device/function-dependent arguments:

- P1—the starting virtual address of the buffer containing the data to be transmitted
- P2—the size of the buffer in bytes

The message size specified by P2 cannot be larger than the maximum send message size for the unit (see Section 1.3). If a message larger than the maximum size is sent, a status of SS$_DATAOVERUN is returned in the I/O status block.

The write functions can take one function modifier:

- IO$M_ENABLMBX—enable use of the associated mailbox

1.4.3 Set Mode

Set mode operations are used to perform protocol, operational, and program/driver interface operations with the DMC11. The VAX/VMS operating system defines five types of set mode functions:

- Set mode
- Set characteristics
- Enable attention AST
- Set mode and shut down unit
- Set mode and start unit

1.4.3.1 Set Mode and Set Characteristics

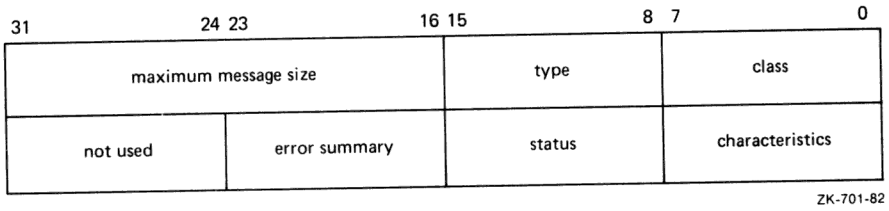
The set mode and set characteristics functions set device characteristics such as maximum message size. The VAX/VMS operating system provides two function codes:

- IO$_SETMODE—set mode (requires logical I/O privilege)
- IO$_SETCHAR—set characteristics (requires physical I/O privilege)

These two functions take the following device/function-dependent argument:

- P1—the virtual address of the quadword characteristics buffer block if the characteristics are to be set. If this argument is zero, only the unit status and characteristics are returned in the I/O status block (see Section 1.5). Figure 1–2 shows the P1 characteristics block.

Figure 1–2 P1 Characteristics Block



In the buffer designated by P1 the device class is DC$_SCOM. Section 1.3 describes the device types. The maximum message size describes the maximum send or receive message size.

The second longword contains device/function-dependent characteristics: unit characteristics, status, and error summary bits. Any of the characteristics values and some of the status values can be set or cleared (see Tables 1–3, 1–4, and 1–5).

If the unit is active (XM$M_STS_ACTIVE is set), the action of a set mode or set characteristics function with a characteristics buffer is to clear the status bits or the error summary bits. If the unit is not active, the status bits or the error summary bits can be cleared, and the maximum message size, type, device class, and unit characteristics can be changed.

1.4.3.2 Enable Attention AST

The enable attention AST function enables an AST to be queued when an attention condition occurs on the unit. An AST is queued when the driver sets or clears either an error summary bit or any of the unit status bits, or when a message is available and there is no waiting read request. The enable attention AST function is legal at any time, regardless of the condition of the unit status bits.

The VAX/VMS operating system provides two function codes:

- IO$_SETMODE!IO$M_ATTNAST—enable attention AST
- IO$_SETCHAR!IO$M_ATTNAST—enable attention AST

Enable attention AST enables an AST to be queued one time only. After the AST occurs, it must be explicitly reenabled by the function before the AST can occur again. The function code is also used to disable the AST. The function is subject to AST quotas.

The enable attention AST functions take the following device/function-dependent arguments:

- P1—address of AST service routine or 0 for disable
- P2—(ignored)
- P3—access mode to deliver AST

The AST service routine is called with an argument list. The first argument is the current value of the device/function-dependent characteristics longword shown in Figure 1–2. The access mode specified by P3 is maximized with the requester's access mode. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume* for an explanation of this concept.)

1.4.3.3 Set Mode and Shut Down Unit

The set mode and shut down unit function stops the operation on an active unit (XM$M_STS_ACTIVE must be set) and then resets the unit characteristics.

The VAX/VMS operating system provides two function codes:

- IO$_SETMODE!IO$M_SHUTDOWN—shut down unit
- IO$_SETCHAR!IO$M_SHUTDOWN—shut down unit

These functions take one device/function-dependent argument:

- P1—the virtual address of the quadword characteristics block (Figure 1–2) if modes are to be set after shutdown. P1 is 0 if modes are not to be set after shutdown.

Both functions stop the DMC11 microprocessor and release all outstanding message blocks; any messages that have not been read are lost. The characteristics are reset after shutdown. Except for the sending of attention ASTs and mailbox messages, these functions act the same as the driver does when shutdown occurs because of a fatal error.

1.4.3.4 Set Mode and Start Unit

The set mode and start unit function sets the characteristics and starts the protocol on the associated unit. The VAX/VMS operating system provides two function codes:

- IO$_SETMODE!IO$M_STARTUP—start unit
- IO$_SETCHAR!IO$M_STARTUP—start unit

These functions take the following device/function-dependent arguments:

- P1—the virtual address of the quadword characteristics block (Figure 1–2) if the characteristics are to be set. Characteristics are set before the device is started.
- P2—(ignored).
- P3—the number of preallocated receive-message blocks to ensure the availability of buffers to receive messages.

The total quota taken from the process's buffered I/O byte count quota is the DMC11 work space plus the number of receive-message buffers specified by P3 times the maximum message size. For example, if six 200-byte buffers are required, the total quota taken is 1456 bytes:

```
   256  (DMC11 work space)
+ 1200  (number of buffers X buffer size)
  ----
  1456  (total quota taken)
```

This quota is returned to the process when shutdown occurs.

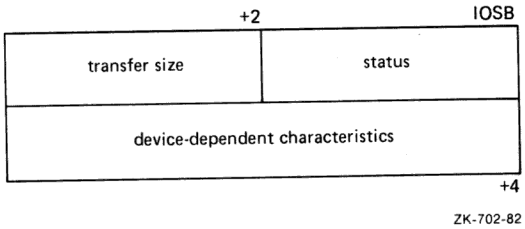
Receive-message blocks are used by the driver to receive messages that arrive independent of read request timing. When a message arrives, it is matched with any outstanding read requests. If there are no outstanding read requests, the message is queued and an attention AST or mailbox message is generated. (IO$_SETMODE!IO$M_ATTNAST or IO$_SETCHAR!IO$M_ATTNAST must be set to enable an attention AST; IO$M_ENABLMBX must be used to enable a mailbox message.)

When read, the receive-message block is returned to the receive-message "free list" defined by P3. If the "free list" is empty, no receive messages are possible. In this case, a data lost condition can be generated if a message arrives. This nonfatal condition is reported by device-dependent data and an attention AST.

1.5 I/O Status Block

The I/O status block (IOSB) usage for all DMC11 functions is shown in Figure 1–3. Appendix A lists the status returns for these functions. (The *VAX/VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.)

Figure 1–3 IOSB Contents



ZK-702-82

In Figure 1–3, the transfer size at IOSB+2 is the actual number of bytes transferred. Table 1–3 lists the device-dependent characteristics returned at IOSB+4. These characteristics can also be obtained by using the Get Device/Volume Information ($GETDVI) system service (see Section 1.3).

1.6 Programming Example

This sample program (Example 1–1) shows the typical use of QIO functions in DMC11/DMR11 driver operations such as transmitting and receiving data and checking for errors.

Example 1–1 DMC11/DMR11 Program Example

```
        .TITLE  EXAMPLE - DMC11/DMR11 Device Driver Sample Program
        .IDENT 'X00'
        $IODEF                  ; Define I/O functions and modes
        $XMDEF                  ; Define driver status flags
;
;  Macro definitions
;
        .macro  type    string,?L               ;
        store   <string>                        ;
        movl    #$$.tmpx,cmdorab+rab$l_rbf      ;
        movw    #$$.tmpx1,cmdorab+rab$w_rsz     ;
        $put    rab=cmdorab                     ;
        blbs    r0,L                            ;
        $exit_s                                 ;
L:                                              ;
        .endm   type                            ;
;
        .macro  store   string,pre
        .save
        .psect  $$$DEV
        $$.tmpx=.
        pre
        .ascii  %string%
        $$.tmpx1=.-$$.tmpx
        .restore
        .endm   store
CMDOFAB:        $FAB    fac=put,fnm=sys$output:,-   ; Output FAB
                mrs=132,rat=cr,rfm=var
CMDORAB:        $RAB    ubf=cmdbuf,usz=cmdbsz,-     ; Output RAB
                fab=cmdofab
CMDBUF::        .BLKB   256                     ; Command buffer
CMDBSZ=         .-CMDBUF                        ; Buffer size
FAOBUFDSC:      .LONG   CMDBSZ,CMDBUF           ; FAO buffer
                                                ; descriptor
FAOLEN:         .BLKL   1                       ; FAO output buffer
                                                ; length
P2BUF::         .BLKL   50                      ; P2 buffer
P2BUFSZ=        .-P2BUF                         ; P2 buffer size
P2BUFDSC:       .LONG   P2BUFSZ,P2BUF           ; P2 buffer descriptor
P1BUF::         .BLKQ   1                       ; P1 buffer
P1BUFSZ=        .-P1BUF                         ; P1 buffer size
CHNL::          .BLKL   1                       ; Channel number
IOSB::          .BLKQ   1                       ; I/O status block
DEVDSC:         .ASCID  'DEV'                   ; Device to assign
QIOREQDSC:      .LONG   QIOREQSZ,QIOREQ         ; QIO request status
QIOREQ:         .ASCII  'QIO completion status = !XL'
                .ASCII  'IOSB1 = !XL, IOSB2 = !XL'
QIOREQSZ=       .-QIOREQ                        ; Size of QIO status
                                                ; report
XMTBUFLEN=512                                   ; Size of transmit
                                                ; buffer
XMTBUF:         .REPEAT XMTBUFLEN
                .BYTE   ^X93                    ; Transmit data
                .ENDR
RCVBUF:         .BLKB   XMTBUFLEN
```

(Continued on next page)

Example 1–1 (Cont.) DMC11/DMR11 Program Example

```
;
; This is the start of the program section.
;
START:: .WORD   0
        $OPEN   FAB=CMDOFAB                     ; Open output
        BLBC    R0,EXIT                         ;
        $CONNECT RAB=CMDORAB                    ; Connect to output
        BLBC    R0,EXIT                         ;
        BRB     CONT                            ; Continue
EXIT:   $EXIT_S                                 ; Exit program
CONT:   TYPE    <DMC11/DMR11 Test Program>
        TYPE    < >
        $ASSIGN_S       DEVNAM=DEVDSC,CHAN=CHNL ; Assign unit
        BLBC    R0,EXIT                         ; Exit on error
;
; Initialize and start controller
;
        MOVZBL  #XM$M_CHR_LOOPB,P1BUF+4         ; Set P1 flags -
                                                ; Loopback
        MOVW    #XMTBUFLEN,P1BUF+2              ; Set P1 buffer size
        CLRL    P2BUFDSC                        ; Set zero length P2
                                                ; buffer
        BSBW    INIT                            ; Issue QIO
;
; Loopback data
;
        MOVZWL  #100,R9                         ; Loop device 100
                                                ; times
10$:    BSBW    XMIT                            ; Issue transmit
        BSBW    RECV                            ; Issue receive
        MOVAB   XMTBUF,R1                       ; Get address of xmit
                                                ; data
        MOVAB   RCVBUF,R2                       ; Get address of
                                                ; received data
        MOVZWL  #XMTBUFLEN,R3                   ; Get number of bytes
                                                ; to verify
20$:    CMPB    (R1)+,(R2)+                     ; Check data
        BNEQ    30$                             ;
        SOBGTR  R3,20$                          ;
        SOBGTR  R9,10$                          ;
        BRW     EXIT                            ; Exit
30$:    TYPE    <*** Loopback buffer comparison error ***>
        BRW     EXIT                            ; Exit
;
; Initialize controller QIO
;
INIT:   TYPE    <*** Initialize controller QIO ***>     ;
        $QIOW_S chan=chnl,func=#io$_setchar!io$m_startup,-
                p1=p1buf,p2=#p2bufdsc,iosb=iosb,p3=#5   ;
        BRW     QIO_STATUS                      ;
```

(Continued on next page)

Example 1–1 (Cont.) DMC11/DMR11 Program Example

```
;
; Xmit data QIO
;
XMIT:   TYPE    <*** Transmit buffer QIO ***>   ;
        $QIO_S  chan=chnl,func=#io$writevblk,p1=xmtbuf,-
                p2=#xmtbuflen,iosb=iosb
        BRW     QIO_XMTST                       ;
;
; Receive data QIO
;
RECV:   TYPE    <*** Receive buffer QIO ***>    ;
        $QIOW_S chan=chnl,efn=#2,func=#io$_readvblk,-
                p1=rcvbuf,p2=#xmtbuflen,iosb=iosb
        .BRB    qio_status
        .ENABL  LSB
QIO_STATUS:
        BLBC    IOSB,10$                        ; Check status of QIO
                                                ; Br if error on QIO
QIO_XMTST:
        BLBC    R0,10$                          ; Check status of XMIT
                                                ; Br if error on
                                                ; request
        RSB                                     ; Else, return to
                                                ; caller
10$:    MOVZWL  IOSB,R1                         ; Get I/O status block
        PUSHL   R1                              ; Push I/O status block
        PUSHL   R0                              ; Push system service
                                                ; status
        PUSHAQ  FAOBUFDSC                       ; Push address of FAO
                                                ; buffer descriptor
        PUSHAW  FAOLEN                          ; Push address of
                                                ; output length
        PUSHAQ  QIOREQDSC                       ; Push address of
                                                ; input string
        CALLS   #5,G^SYS$FAO                    ; Get error message
        MOVAB   CMDBUF,CMDORAB+RAB$L_RBF        ; Get output buffer
                                                ; address
        MOVW    FAOLEN,CMDORAB+RAB$W_RSZ        ; Get output buffer
                                                ; length
        $PUT    CMDORAB                         ; Print error text
        BRW     EXIT                            ; Exit
        .DSABL  LSB
        .END    START
```

2 DMP11, DMF32, and Asynchronous DDCMP Interface Drivers

This section describes the use of the VAX/VMS DMP11 multipoint communications line interface, DMF32 synchronous line interface, and asynchronous DDCMP interface drivers.

2.1 Supported Devices

The DMP11 multipoint communications line interface is a direct-memory-access (DMA) device that uses the DIGITAL Data Communications Message Protocol (DDCMP) to provide direct communication between a VAX processor and DDCMP-compatible devices, such as other DMP11s and some terminals (for example, the VT62). Up to 32 devices can be connected to the DMP11 through a single, multidrop, DDCMP-compatible line.

The logical connection between the DMP11 and a connected device is called a *tributary*. In multipoint configurations, the DMP11 functions as a multipoint control station, and the devices on the DDCMP line are located at tributary addresses. A controller operating in tributary mode on this line is called a *tributary station*.

In point-to-point configurations, one DMP11 is connected to one other controller. Controllers in this mode are called *point-to-point stations*.

The DMF32 synchronous line interface is a DMA communications device that uses a software implementation of DDCMP to provide an interface between a VAX processor and other DDCMP-compatible devices, such as a DMP11 or DMC11. The DMF32 supports both full- and half-duplex modes as well as tributary mode on a multidrop DDCMP-compatible line.

In a multipoint configuration, the DMF32 operates in tributary mode and is located at a tributary address on the DDCMP line.

In point-to-point configurations, one DMF32 is connected to a single other controller. Controllers in this mode are called point-to-point stations.

Asynchronous DDCMP is supported for DECnet–VAX using software DDCMP over terminal ports. This enables all VAX/VMS-supported terminal devices to provide a DDCMP interface between two VAX processors using terminal ports. Asynchronous DDCMP supports full-duplex, point-to-point lines.

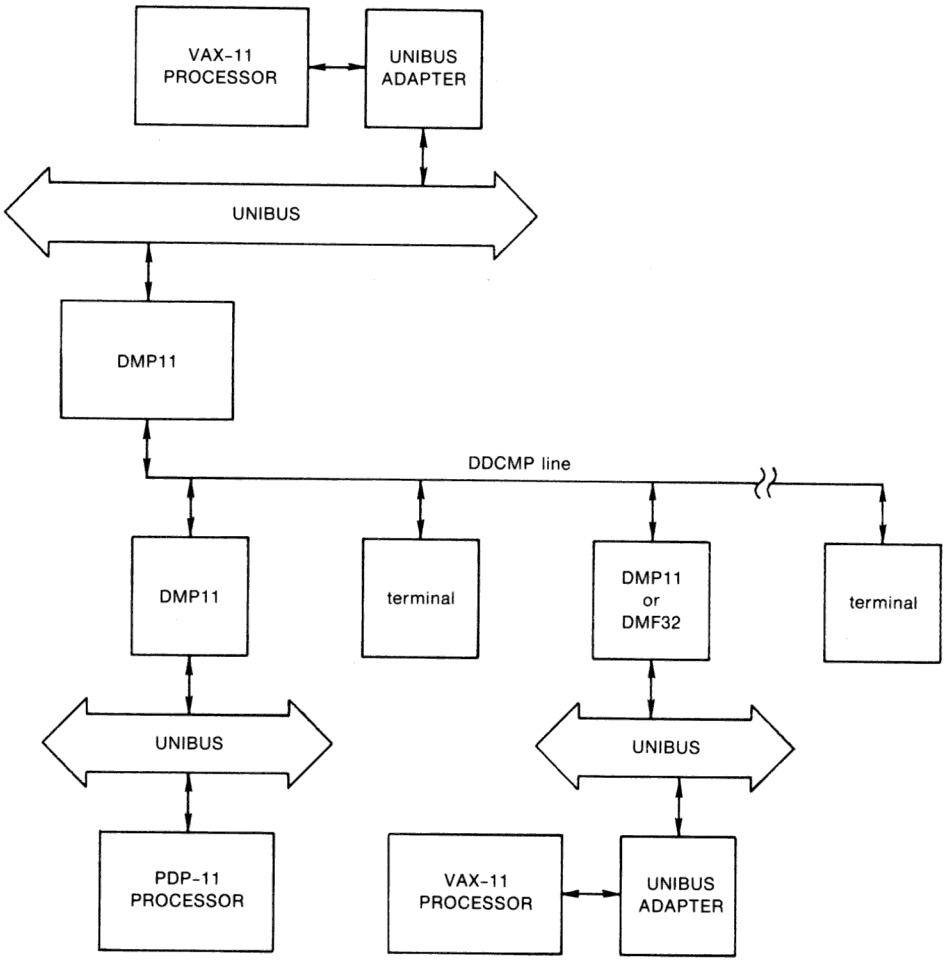
Figure 2–1 shows a typical DMP11/DMF32 multipoint configuration.

2.2 Driver Features and Capabilities

The DMP11, DMF32, and asynchronous DDCMP drivers provide the following capabilities:

- Multipoint operating mode in which the DMP11 functions as a control station connected to from 1 to 32 devices and tributary stations (not for the DMF32 or asynchronous DDCMP drivers)

Figure 2–1 Typical DMP11/DMF32 Multipoint Configuration



- Multipoint operating mode in which the DMP11 or DMF32 functions as a tributary station (not for the asynchronous DDCMP driver)
- Point-to-point operating mode in which the DMP11, DMF32, or asynchronous DDCMP port is connected to a single other controller also operating in point-to-point mode
- DMC11-compatible operating mode in which the DMP11 is connected to either a DMC11, a DMR11, another synchronous line interface using DDCMP, or another DMP11 running in DMC11-compatible mode (not for the DMF32 or asynchronous DDCMP drivers)
- Support for using the DMF32 in high-level data link control (HDLC) bit stuff mode
- Support for using a general character-oriented protocol over the DMF32
- A nonprivileged QIO interface to the DMP11, DMF32, and asynchronous DDCMP for using these devices as raw-data channels
- Tributary attention conditions transmitted through attention ASTs

- Full- and half-duplex operation (full duplex only with the asynchronous DDCMP driver)
- Interface design common to all communications devices supported by the VAX/VMS operating system
- Separate transmit and receive queues
- Assignment of multiple read and write buffers to the device

2.2.1 Character-Oriented Protocols and HDLC Bit Stuff Mode

The DMF32 synchronous line unit supports character-oriented protocols and the high-level data link control (HDLC) bit stuff mode. The DMF32 driver can transmit and receive a framed message and also provide some modem control. General protocol handling for the character-oriented protocols is supported at the DMF32 driver level. However, the DMF32 driver will provide an interface to the higher-level protocol so that receive messages will be framed by the rules of the protocol. For HDLC mode, the user is provided with a method to transmit and receive frame messages in full-duplex mode only.

Sections 2.4.3.2 through 2.4.3.5 describe these features of the DMF32 driver in greater detail.

2.2.2 Quotas

Transmit operations are direct (DMP11) or buffered (DMF32 and asynchronous DDCMP) I/O operations and are limited by the process's direct or buffered I/O quota.

The quotas for the receive buffer free list (see Section 2.4.3.1) are the process's buffered I/O quota and buffered I/O byte count quota.

2.2.3 Power Failure

If a system power failure occurs, no DMP11, DMF32, or asynchronous DDCMP recovery is possible. The driver is in a fatal error state and shuts down.

2.3 Device Information

Users can obtain information on DMP11, DMF32, or asynchronous DDCMP characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume*.)

$GETDVI returns device characteristics when you specify the item code DVI$_DEVCHAR. Table 2–1 lists these characteristics, which are defined by the $DEVDEF macro.

DVI$_DEVCLASS returns the device class, which is DC$_SCOM for the DMP11 and the DMF32. DVI$_DEFTYPE returns the device type, which is DT$_DMP11 for the DMP11 and DT$_DMF32 for the DMF32. The $DCDEF macro defines the device class and device type names.

DVI$_DEVBUFSIZ returns the maximum message size. The maximum message size is the maximum send or receive message size for the unit. Messages greater than 512 bytes on modem-controlled lines are more prone to transmission errors.

Table 2–1 DMP11 and DMF32 Device Characteristics

Characteristic[1]	Meaning
Static Bits (Always Set)	
DEV$M_NET	Network device. Set for terminal port if it is a network device.
DEV$M_AVL	Available device. Set when unit control block (UCB) is initialized.
DEV$M_ODV	Output device.
DEV$M_IDV	Input device.
DEV$M_SHR[2]	Shareable device.

[1]Defined by the $DEVDEF macro

[2]Only for DMP11

DVI$_DEVDEPEND returns the unit characteristics bits, the unit and line status bits, the error summary bits, and the specific error(s) in a longword field. (Byte 0 = characteristics, byte 1 = status, byte 2 = error summary, and byte 3 = specific error(s).)

Unit characteristics bits govern the DDCMP operating mode. They are defined by the $XMDEF macro and can be set by a set mode function (see Section 2.4.3.1) or can be read by a sense mode function (see Section 2.4.4). Table 2–2 lists the unit characteristics values and their meanings.

Table 2–2 DMP11 and DMF32 Unit Characteristics

Characteristic	Meaning
XM$M_CHR_MOP	Specifies DDCMP maintenance mode
XM$M_CHR_LOOPB[2]	Specifies loopback mode
XM$M_CHR_HDPLX[2]	Specifies half-duplex operation
XM$M_CHR_CTRL[1,2]	Specifies control station
XM$M_CHR_TRIB[2]	Specifies tributary station
XM$M_CHR_DMC[1,2]	Specifies DMC11-compatible mode

[1]Only for DMP11

[2]Not supported for asynchronous DDCMP

The status bits show the status of the unit and the line. These bits can only be set or cleared when the controller and tributary are not active.

Table 2–3 lists the status values and their meanings. The values are defined by the $XMDEF macro.

Table 2–3 DMP11 and DMF32 Unit and Line Status

Status	Meaning
XM$M_STS_ACTIVE	DDCMP protocol is active.
XM$M_STS_DISC	Modem line went from on to off. This bit will be returned in the field IRP$L_IOST2 if the driver has had a timeout while waiting for the CTS signal to be present on the device.
XM$M_STS_RUNNING[1]	Tributary is responding.
XM$M_STS_BUFFAIL	Receive buffer allocation failed.

[1]Only for DMP11

The error summary bits are set when an error occurs. They are read-only bits. If the error is fatal, the DMP11 or DMF32 is shut down. Table 2–4 lists the error summary bit values and their meanings.

Table 2–4 Error Summary Bits

Error Summary Bit	Meaning
XM$M_ERR_MAINT	DDCMP maintenance message received
XM$M_ERR_START	DDCMP start message received
XM$M_ERR_FATAL	Hardware or software error occurred on controller
XM$M_ERR_TRIB	Hardware or software error occurred on tributary
XM$M_ERR_LOST	Data lost when a received message was longer than the specified maximum message size
XM$M_ERR_THRESH	Receive, transmit, or select threshold errors

Table 2–5 lists the errors that can be specified. These errors are mapped to the indicated codes.

Table 2–5 DMP11 and DMF32 Errors

Value[1] (octal)	Meaning	Code Set
2	Receive threshold error	XM$M_ERR_THRESH
4	Transmit threshold error	XM$M_ERR_THRESH
6	Select threshold error	XM$M_ERR_THRESH
10	Start received in run state	XM$M_ERR_START
12	Maintenance received in run state	XM$M_ERR_MAINT
14	Maintenance received in halt state	(none)
16	Start received in maintenance state	XM$M_ERR_START
22	Dead tributary	XM$M_STS_RUNNING[2] (cleared)

[1]Not provided on the DMF32 or asynchronous DDCMP

[2]Not supported for the DMF32 or asynchronous DDCMP

Table 2–5 (Cont.) DMP11 and DMF32 Errors

Value[1] (octal)	Meaning	Code Set
24	Running tributary	XM$M_STS_RUNNING[2] (set)
26	Babbling tributary	XM$M_ERR_TRIB
30	Streaming tributary	XM$M_ERR_TRIB
32	Ring detection	(none)
100–276	Internal procedure (software) errors	XM$M_ERR_TRIB
300	Buffer too small	XM$M_ERR_LOST
302	Nonexistent memory	XM$M_ERR_FATAL
304	Modem disconnected	XM$M_STS_DISC and XM$M_ERR_FATAL[3]
306	Queue overrun	XM$M_ERR_FATAL[2]
310	Carrier lost on modem	XM$M_ERR_FATAL[3]

[1]Not provided on the DMF32 or asynchronous DDCMP

[2]Not supported for the DMF32 or asynchronous DDCMP

[3]Not supported for asynchronous DDCMP

2.4 DMP11, DMF32, and Asynchronous DDCMP Function Codes

The DMP11, DMF32, and asynchronous DDCMP drivers can perform logical, virtual, and physical I/O operations. The basic functions are read, write, set mode, set characteristics, and sense mode. Table 2–6 lists these functions and their function codes. The sections that follow describe these functions in greater detail.

Table 2–6 DMP11, DMF32, and Asynchronous DDCMP I/O Functions

Function Code and Arguments	Type[1]	Modifiers	Function
IO$_READLBLK P1,P2	L	IO$M_NOW	Read logical block.
IO$_READVBLK P1,P2	V	IO$M_NOW	Read virtual block.
IO$_READPBLK P1,-P2,[P6]	P	IO$M_NOW	Read physical block.
IO$_WRITELBLK P1,P2	L		Write logical block.
IO$_WRITEVBLK P1,P2	V		Write virtual block.
IO$_WRITEPBLK P1,-P2,[P6]	P		Write physical block.

[1]V = virtual, L = logical, P = physical (There is no functional difference in these operations.)

Table 2–6 (Cont.) DMP11, DMF32, and Asynchronous DDCMP I/O Functions

Function Code and Arguments	Type[1]	Modifiers	Function
IO$_SETMODE P1,- [P2],P3	L	IO$M_CTRL IO$M_SHUTDOWN IO$M_STARTUP IO$M_ATTNAST IO$M_SET_MODEM[2]	Set DMP11, DMF32, and asynchronous DDCMP characteristics and controller state for subsequent operations.
IO$_SETCHAR P1,- [P2],P3,[P6]	P	IO$M_CTRL IO$M_SHUTDOWN IO$M_STARTUP IO$M_ATTNAST IO$M_SET_MODEM[2]	Set DMP11, DMF32, and asynchronous DDCMP characteristics and controller state for subsequent operations.
IO$_SENSEMODE P1,P2	L	IO$M_CTRL IO$M_RD_MEM[2] IO$M_RD_MODEM[3] IO$M_RD_COUNT[2] IO$M_CLR_COUNT	Sense controller or tributary characteristics and return them in specified buffer(s).
IO$_CLEAN	L		Complete outstanding requests (character-oriented protocols), and abort outstanding transmits (bit stuff mode).

[1]V = virtual, L = logical, P = physical (There is no functional difference in these operations.)

[2]Only for DMP11

[3]Not for asynchronous DDCMP

Although the DMP11, DMF32, and asynchronous DDCMP drivers do not differentiate among logical, virtual, and physical I/O functions (all are treated identically), the user must have the required privilege to issue a request.

2.4.1 Read

Read functions provide for the direct transfer of data into the user process's virtual memory address space. The VAX/VMS operating system provides three function codes:

- IO$_READLBLK—read logical block
- IO$_READVBLK—read virtual block
- IO$_READPBLK—read physical block

Received messages are multibuffered in system dynamic memory and then copied to the user's buffer.

The read functions take the following device/function-dependent arguments:

- P1—the starting virtual address of the buffer that is to receive data
- P2—the size of the receive buffer in bytes
- P6—the address of a diagnostic buffer; only for physical I/O functions (optional); not supported for asynchronous DDCMP operations (See Section 2.4.5.)

The message size specified by P2 cannot be larger than the maximum receive-message size for the unit (see Section 2.3). If a message larger than the maximum size is received, a status of SS$_DATAOVERUN is returned in the I/O status block.

The read functions can take one function modifier:

- IO$M_NOW—complete the read operation immediately with a received message (If no message is currently available, return a status of SS$_ENDOFFILE in the I/O status block.)

2.4.2 Write

Write functions provide for the direct transfer of data from the user process's virtual memory address space. The VAX/VMS operating system provides three function codes:

- IO$_WRITELBLK—write logical block
- IO$_WRITEVBLK—write virtual block
- IO$_WRITEPBLK—write physical block

Transmitted DMP11 messages are sent directly from the requesting process's buffer. DMF32 and asynchronous DDCMP messages are copied into a system buffer before they are transmitted.

The write functions take the following device/function-dependent arguments:

- P1—the starting virtual address of the buffer containing the data to be transmitted
- P2—the size of the buffer in bytes

- P6—the address of a diagnostic buffer; only for physical I/O functions (optional); not supported for asynchronous DDCMP operations (See Section 2.4.5.)

The message size specified by P2 cannot be larger than the maximum send-message size for the unit (see Section 2.3).

The write functions take no function modifiers.

2.4.3 Set Mode and Set Characteristics

Set mode operations are used to perform protocol, operational, and program/driver interface operations with the DMP11, DMF32, and asynchronous DDCMP drivers. The VAX/VMS operating system defines seven types of set mode functions:

- Set mode
- Set characteristics
- Set controller mode
- Set tributary mode
- Enable attention AST
- Shutdown controller
- Shutdown tributary

Used without function modifiers, set mode and set characteristics functions can modify an existing tributary. Used with certain function modifiers, they can perform DMP11, DMF32, and asynchronous DDCMP operations such as starting a tributary and requesting an attention AST. The VAX/VMS operating system provides two function codes:

- IO$_SETMODE—set mode (requires logical I/O privilege)
- IO$_SETCHAR—set characteristics (requires physical I/O privilege)

The other five types of set mode functions, which use the two function codes with certain function modifiers, are described in the sections that follow.

To use the IO$_SETMODE and IO$_SETCHAR functions, the user must assign the appropriate unit control block (UCB) with the Assign I/O Channel ($ASSIGN) system service.

2.4.3.1 Set Controller Mode

The set controller mode function sets the DMP11, DMF32, or asynchronous DDCMP controller state and activates the controller. For asynchronous DDCMP operations, the first occurrence of an IO$_SETMODE function creates a buffer for the driver to use. (Part of the buffer created by IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP is allocated for the protocol operation to use.) Four combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_CTRL—set controller characteristics
- IO$_SETCHAR!IO$M_CTRL—set controller characteristics
- IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP—set controller characteristics and start the controller

- IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP—set controller characteristics and start the controller

If the function modifier IO$M_STARTUP is specified, the controller is started and the modem is enabled. If IO$M_STARTUP is not specified, the specified characteristics are simply modified.

These codes take the following device/function-dependent arguments:

- P1—the virtual address of a quadword characteristics buffer
- P2—the address of a descriptor for an extended characteristics buffer (optional)
- P3—the number of preallocated receive-message blocks to allocate (referred to as the size of the "common receive pool") (See the NMA$C_PCLI_BFN parameter ID in Table 2–8.)

Figure 2–2 shows the format of the P1 characteristics buffer. The maximum message size in the first longword specifies the maximum allowable transmit and receive-message length.

Table 2–7 lists the DMP11 and DMF32 characteristics that can be set in the second longword. The $XMDEF macro defines these values. No asynchronous DDCMP characteristics can be set using the P1 characteristics buffer.

The P2 buffer consists of a series of six-byte entries. The first word contains the parameter identifier (ID), and the longword that follows contains one of the values that can be associated with the parameter ID. Figure 2–3 shows the format for this buffer.

If both P1 and P2 characteristics are specified, the P2 characteristics supersede the P1 characteristics. For example, if P1 specifies XM$M_CHR_CTRL and P2 specifies NMA$C_PCLI_PRO with a value of NMA$C_LINPR_TRIB (that is, a tributary), the device will be started as a tributary.

Figure 2–2 P1 Characteristics Buffer (Set Controller)

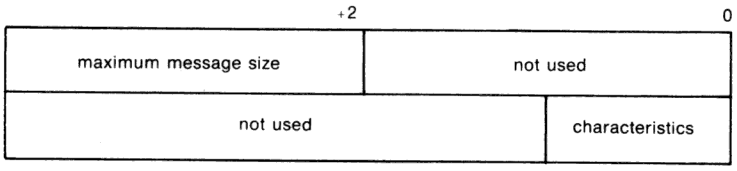


Table 2–7 DMP11 and DMF32 Characteristics

Characteristic[1]	Meaning
XM$M_CHR_LOOPB	Sets loopback mode
XM$M_CHR_HDPLX	Sets half-duplex operation
XM$M_CHR_CTRL[2]	Specifies control station
XM$M_CHR_TRIB	Specifies tributary station
XM$M_CHR_DMC[2]	Specifies DMC11-compatible mode

[1]Not supported for asynchronous DDCMP

[2]Only for DMP11

Figure 2–3 P2 Extended Characteristics Buffer

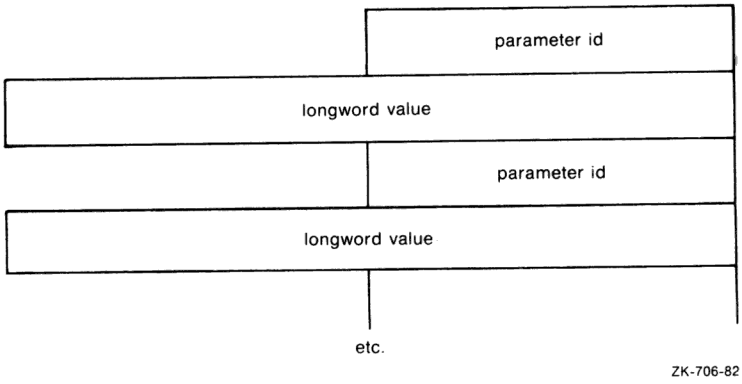


Table 2–8 lists the parameter IDs and values that can be specified in the P2 buffer. The $NMADEF macro defines these values.

Section 2.4.3.2 lists the parameter IDs allowed for the character-oriented and HDLC bit stuff modes of operation.

Table 2–8 P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_PRO	Protocol mode. The following values can be specified:

Value	Meaning
NMA$C_LINPR_POI	DDCMP point-to-point (default)
NMA$C_LINPR_CON[1]	DDCMP control station
NMA$C_LINPR_TRI[2]	DDCMP tributary
NMA$C_LINPR_DMC[1]	DDCMP DMC mode
NMA$C_LINPR_LAPB[3]	HLDC bit stuff mode
NMA$C_LINPR_BSY[3]	General character-oriented protocol mode

NMA$C_PCLI_DUP	Duplex mode. The following values can be specified:

Value	Meaning
NMA$C_DPX_FUL	Full-duplex (default)
NMA$C_DPX_HAL[2]	Half-duplex

NMA$C_PCLI_CON	Controller mode. The following values can be specified:

Value	Meaning
NMA$C_LINCN_NOR	Normal (default)
NMA$C_LINCN_LOO[2]	Loopback

NMA$C_PCLI_BFN	Number of receive buffers to preallocate. Must be provided here or as P3 argument.
NMA$C_PCLI_BUS	Maximum allowable transmit and receive message length (default = 512 bytes).
NMA$C_PCLI_NMS	Number of sync characters to precede message.
NMA$C_PCLI_SLT[1,4]	Number of milliseconds (msec) in the period of incrementing tributary priorities and the transmit delay (min = 50; default = 50).
NMA$C_PCLI_DDT[1,4]	Number of msec in the period of polling dead tributaries (default = 10000).
NMA$C_PCLI_DLT[1,4]	Number of msec between polls (default = 0).
NMA$C_PCLI_SRT[1,4]	Timer value used by control station and half-duplex point-to-point to establish that a tributary is streaming (default = 6000).

[1]Only for DMP11

[2]Not for asynchronous DDCMP

[3]Only for DMF32

[4]A global polling parameter. All timer values must be specified in milliseconds.

2.4.3.2 Additional Features of the DMF32 Driver

The character-oriented protocols and the HDLC bit stuff mode do not have the concept of line and circuit. Therefore, only $QIO requests that include the function modifier IO$M_CTRL are allowed. The VAX/VMS operating system does not acknowledge the characteristics set in the P1 buffer for the character-oriented and HDLC bit stuff modes of operation. Note that users must have CMKRNL privilege to run the DMF32 in character-oriented mode. Only the parameters listed in Table 2–9 are relevant to the character-oriented and HDLC bit stuff modes of operation:

Table 2–9 P2 Extended Characteristics Values (DMF32 Driver)

Parameter ID	Meaning
NMA$C_PCLI_PRO	This parameter must be set to NMA$C_LINPR_BSY to specify character-oriented mode of operation, or to NMA$C_LINPR_LAPB to specify HDLC bit stuff mode.
NMA$C_PCLI_DUP	This parameter requests full- or half-duplex mode of operation. (HDLC bit stuff mode supports full-duplex mode only.) If half-duplex mode is specified, the DMF32 driver sets the request to send (RTS) signal, waits for the clear to send (CTS) signal at the beginning of the transmit, and then drops RTS at the end of the transmit. The full-duplex mode value is NMA$C_DPX_FUL; the half-duplex mode value is NMA$C_DPX_HAL.
NMA$C_PCLI_BFN	The number of buffers the device can allocate for use as receive buffers. This value must be greater than 1. Default is 4.
NMA$C_PCLI_BUS	The size of the buffers to be allocated.
NMA$C_PCLI_CON	The state the controller is set to. If NMA$C_LINCN_NOR is specified, the device operates normally. If NMA$C_LINCN_LOO is specified, the device operates in internal loopback mode. Default is normal operation.
NMA$C_PCLI_SYC[1]	The sync character used by device. Defaults to 32 hexadecimal.
NMA$C_PCLI_NMS[1]	The number of sync characters to precede a transmit. Defaults to 8.
NMA$C_PCLI_BPC[1]	The number of bits per character (5,6,7, or 8). Defaults to 8.
NMA$C_PCLI_FRA[1]	The address of the protocol framing routine (in nonpaged pool). This parameter must be specified.
NMA$C_PCLI_STI1[1] NMA$C_PCLI_STI2[1]	These two parameters contain the initial value for the quadword of framing routine state information.

[1] Character-oriented mode only

Table 2–9 (Cont.) P2 Extended Characteristics Values (DMF32 Driver)

Parameter ID	Meaning
NMA$C_PCLI_MCL[1]	Determines whether modem signals should be turned off when a DEASSIGN operation is performed. The DMF32 driver always clears the modem signals on the last DEASSIGN. However, on all other DEASSIGN operations the modem signals will only be cleared if the value of NMA$C_PCLI_MCL is 0. If the value NMA$C_STATE_ON is specified, the data terminal ready (DTR) signal is dropped when DEASSIGN is performed. If the value NMA$C_STATE_OFF specified, DTR is not dropped until the last DEASSIGN.
NMA$C_PCLI_TMO[1]	Specifies the timeout (in seconds) when waiting for CTS during transmit operations.

[1]Character-oriented mode only

2.4.3.3 Framing Routine Interface for Character-Oriented Protocols

In general, the character-oriented protocols each have their own rules for framing receive messages. To provide support for each protocol's special framing rules, the DMF32 driver has been extended to provide support for calling a special framing routine from the DMF32 driver's processing of receive messages. This routine is defined by the higher-level software using the DMF32 driver and is loaded by that same software into nonpaged pool. The address of this routine is passed to the driver when the device is started up. The purpose of the framing routine is to tell the driver how to frame each byte of the received data message and to tell the driver that the received message is complete and ready to be posted.

The address of the framing routine is kept in the DMF32 driver's internal buffer. The internal buffer also contains a quadword that is used by the framing routine for holding state information while it is framing the receive message. The framing routine is called by the driver at FORK IPL through a JSB instruction. The input and the output to the framing routine is described in the following tables.

Input	Contents
R0	Address of quadword of state information.
R1 bits 0-7	Character to examine. The high-order bit is set if this is the first character of a new frame.

Output	Contents
R0	Status information for the DMF32 driver. The following bits are defined:

Value	Meaning
XG$V_BUFFER_CHAR	If clear, buffer the character in the next position. If set, use bit XG$V_BUFFER_IN_PREV_POS.
XG$V_BUFFER_IN_PREV_POS	If clear, ignore the character. If set, buffer the character in the previous position; do not update the buffer pointer.
XG$V_COMPLETE_READ	If clear, ignore. If set, return the framed buffer to user (buffer character if required).

Note that after the DMF32 driver has completed a framed receive data message, the driver resets the quadword of state information to the value passed when the device is started up. This means that the driver resets error information along with success information.

2.4.3.4 Changes to the DMF32 Driver Transmitter Interface for Character-Oriented Mode

For write requests made through the QIO interface, the P4 parameter will contain the address of a quadword buffer to be used to update the field in the DMF32 driver's internal buffer, which contains the state information for the framing routine. If this parameter is 0, the state information is not updated.

The DMF32 driver interface is also changed in the way errors are returned. The bit XM$M_STS_DISC will be returned in the field IRP$L_IOST2 if the driver has had a timeout while waiting for the CTS signal to be present on the device.

2.4.3.5 The IO$_CLEAN Function

The clean function either completes or aborts outstanding device requests. The VAX/VMS operating system provides the following function code:

- IO$_CLEAN

For character-oriented protocols, a clean function request results in the completion of all outstanding I/O requests pending on the device. For HDLC bit stuff mode, a clean function request results in the aborting of all outstanding transmit operations on the device. In both cases the status return is SS$_ABORT. Note that the modem registers are not cleared.

The clean function is not supported in DDCMP mode of operation.

2.4.3.6 Set Tributary Mode

The set tributary mode function either starts a tributary or modifies an existing one. The driver creates a circuit data block for a particular unit of the DMP11 device with the specified tributary address. The set tributary function must be performed before any communication can occur with the attached unit.

Because the DMF32 and asynchronous DDCMP drivers deal with only one tributary, the set tributary function starts both the tributary and the protocol. The data block that describes the tributary has already been created.

The VAX/VMS operating system provides four combinations of function code and modifier:

- IO$_SETMODE—modify tributary characteristics
- IO$_SETCHAR—modify tributary characteristics
- IO$_SETMODE!IO$M_STARTUP—start tributary
- IO$_SETCHAR!IO$M_STARTUP—start tributary

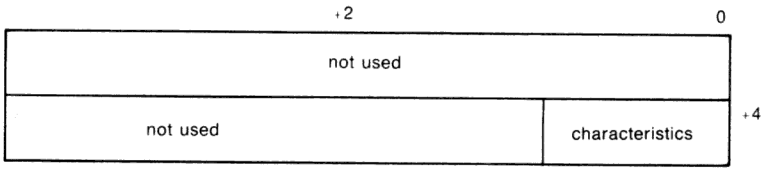
These codes take the following device/function-dependent arguments:

- P1—the virtual address of a quadword characteristics buffer (optional)
- P2—the address of a descriptor for an extended characteristics buffer (optional)

Figure 2–4 shows the format of the P1 characteristics buffer. The following characteristic can be set in the second longword:

- XM$V_CHR_MOP—set tributary to DDCMP maintenance mode

Figure 2–4 P1 Characteristics Buffer (Set Tributary)



The P2 buffer consists of a series of six-byte entries. The first longword contains the parameter identifier (ID), and the longword that follows contains one of the values that can be associated with the parameter ID. Figure 2–3 shows the format for this buffer.

Table 2–10 lists the parameter IDs and values that can be specified in the P2 buffer.

Table 2–10 P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCCI_TRI	Tributary address. Because the maximum physical address that the DMP11 or DMF32 can recognize is 255, only the first byte is actually used. For the DMP11 this parameter must be set before the tributary is started, unless the controller was set to run in point-to-point or DMC-compatible mode. For the DMF32 the tributary address always defaults to 1. Accepted values are 1 to 255.
NMA$C_PCCI_MRB[1]	Maximum number of buffers allocated from common pool for receive messages; 255 indicates unlimited number (default is unlimited). Accepted values are 1 to 255.
NMA$C_PCCI_MST[1]	Maintenance state. The following values can be specified:

Value	Meaning
NMA$C_STATE_ON	On
NMA$C_STATE_OFF	Off (default)

NMA$C_PCCI_POL[1,2]	Latch polling state. The following values can be specified:

Value	Meaning
NMA$C_CIRPST_AUT	Automatic (default)
NMA$C_CIRPST_ACT	Active
NMA$C_CIRPST_INA	Inactive
NMA$C_CIRPST_DIE	Dying
NMA$C_CIRPST_DED	Dead

NMA$C_PCCI_TRT[1,2]	Transmit delay timer (default = 0).
NMA$C_PCCI_ACB[1,2]	Initial poll priority for active state of tributary (default = 255).
NMA$C_PCCI_ACI[1,2]	Rate of priority incrementing for active state of tributary (default = 0).
NMA$C_PCCI_IAB[1,2]	Initial poll priority for inactive state of tributary (default = 0).
NMA$C_PCCI_IAI[1,2]	Rate of priority incrementing for inactive state of tributary (default = 64).
NMA$C_PCCI_DYB[1,2]	Initial poll priority for dying state of tributary (default = 0).
NMA$C_PCCI_DYI[1,2]	Rate of priority incrementing for dying state of tributary (default = 16).
NMA$C_PCCI_IAT[1,2]	Number of no data message responses before changing state to inactive (default = 8).

[1]Only for the DMP11

[2]A tributary-specific polling parameter (All timer values must be specified in milliseconds.)

Table 2–10 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCCI_DYT[1,2]	Number of no responses before changing state to dying (default = 2).
NMA$C_PCCI_DTH[1,2]	Number of no responses before changing state to dead (default = 16).
NMA$C_PCCI_MTR[2]	Maximum number of abutting data messages that will be transmitted before deselecting the tributary (default = 4).
NMA$C_PCCI_BBT[1,2]	Timer value for tributary to indicate maximum amount of time for a selected tributary to transmit. If this value is exceeded, the tributary is babbling (default = 6000).
NMA$C_PCCI_RTT[2]	Retransmit timer for full-duplex point-to-point mode and selection timer for multipoint control and half-duplex point-to-point mode (default = 3000).

[1]Only for the DMP11

[2]A tributary-specific polling parameter (All timer values must be specified in milliseconds.)

If both P1 and P2 characteristics are specified, the P2 characteristics supersede the P1 characteristics. For example, if P1 specifies XM$M_CHR_MOP and P2 specifies NMA$C_PCCI_MST with a value of NMA$C_STATE_OFF, the tributary is in the normal DDCMP or data mode.

On receipt of the QIO request, the DMP11 driver verifies that a tributary address has been specified and determines whether this address is currently in use. If the address is in use, the tributary is not restarted. However, modifications to the tributary state or polling parameters are performed. If the tributary does not already exist, a new tributary is started.

On receipt of the QIO request to a DMF32 or for asynchronous DDCMP, the driver modifies the tributary parameters and starts the protocol. The tributary state and the protocol state are equal. The driver does not verify that a tributary address has been provided. If an address has not been provided, it defaults to 1.

2.4.3.7 Shutdown Controller

The shutdown controller function shuts down the controller and disables the modem line. On completion of a shutdown controller request, all tributaries have been halted (including those tributaries not explicitly halted), all tributary buffers returned, and the controller reinitialized. For the DMF32 and asynchronous DDCMP, this function halts the tributary, the protocol, and the line. The controller cannot be used again until another IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP or IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP request has been issued (see Section 2.4.3.1).

The VAX/VMS operating system provides two combinations of function code and modifier:

- IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN—shutdown controller
- IO$_SETCHAR!IO$M_CTRL!IO$M_SHUTDOWN—shutdown controller

The shutdown controller function takes no device/function-dependent arguments.

2.4.3.8 Shutdown Tributary

The shutdown tributary function halts, but does not delete, the specified tributary. On completion of a shutdown tributary request, the tributary is halted, all buffers are returned, and all pending I/O requests and received messages are aborted. Although the tributary cannot be used again until another IO$_SETMODE!IO$M_STARTUP or IO$_SETCHAR!IO$M_STARTUP request has been issued (see Section 2.4.3.2), all previously defined tributary parameters remain set (applicable only to the DMP11). For the DMF32 and asynchronous DDCMP, this function halts the tributary and the protocol. The attached device cannot be used until the tributary is restarted.

The VAX/VMS operating system provides two combinations of function code and modifier:

- IO$_SETMODE!IO$M_SHUTDOWN—shutdown tributary
- IO$_SETCHAR!IO$M_SHUTDOWN—shutdown tributary

The shutdown tributary function takes no device/function-dependent arguments.

2.4.3.9 Enable Attention AST

The enable attention AST function requests that an attention AST be delivered to the requesting process when a status change occurs on the specified tributary. An AST is queued when the driver sets or clears either an error summary bit or any of the unit status bits (see Tables 2–3 and 2–4), or when a message is available and there is no waiting read request. The enable attention AST function is legal at any time, regardless of the condition of the unit status bits.

The VAX/VMS operating system provides two combinations of function code and modifier:

- IO$_SETMODE!IO$M_ATTNAST—enable attention AST
- IO$_SETCHAR!IO$M_ATTNAST—enable attention AST

These codes take the following device/function-dependent arguments:

- P1—the address of an AST service routine or 0 for disable
- P2—(ignored)
- P3—access mode to deliver AST

The enable attention AST function enables an attention AST to be delivered to the requesting process once only. After the AST occurs, it must be explicitly reenabled by the function before the AST can occur again. The function is also subject to AST quotas.

The AST service routine is called with an argument list. The first argument is the current value of the second longword of the I/O status block (see Section 2.5). The access mode specified by P3 is maximized with the requester's access mode.

2.4.4 Sense Mode

The sense mode function returns the controller or tributary characteristics in the specified buffer(s).

The VAX/VMS operating system provides two function codes:

- IO$_SENSEMODE!IO$M_CTRL—read controller characteristics
- IO$_SENSEMODE—read tributary characteristics

These codes take the following device/function-dependent arguments:

- P1—the address of a two-longword buffer into which the device characteristics are stored (optional). (Figure 2–2 shows the characteristics buffer for controllers; Figure 2–4 shows the characteristics buffer for tributaries.)
- P2—the address of a descriptor for a buffer into which the extended characteristics buffer is stored (optional). (Figure 2–3 shows the format of the extended characteristics buffer.)

All characteristics that fit into the buffer specified by P2 are returned. However, if all the characteristics cannot be stored in the buffer, the I/O status block returns the status SS$_BUFFEROVF. The second word of the I/O status block returns the size (in bytes) of the extended characteristics buffer returned by P2 (see Section 2.5).

2.4.5 Diagnostic Support

The DMP11 and DMF32 drivers provide special capabilities for diagnostic support. The sections that follow describe these capabilities.

If a diagnostic buffer (P6) is specified with a physical I/O request, the eight one-byte device registers are dumped into it on completion of the request. (The DMF32 returns five one-word device registers.) The *DMP11 Technical Manual* and the *DMF32 Technical Manual* specify the contents of these registers. The P6 buffer does not return error counters.

2.4.5.1 Set Line Unit Modem Status

The set line unit modem status function sets the DMP11's line unit modem register. It is not supported for the DMF32. The VAX/VMS operating system provides two combinations of function code and modifier:

- IO$_SETMODE!IO$M_SET_MODEM—set line unit modem status
- IO$_SETCHAR!IO$M_SET_MODEM—set line unit modem status

These codes take the following device/function-dependent argument:

- P1—the address of a longword buffer that contains new modem status. One or more of the symbolic offsets listed in the following table can be set in the buffer.

Offset	Meaning
XM$V_MDM_STNDBY	Select standby used with EIA modems
XM$V_MDM_MAINT2	Maintenance mode 2 for remote loopback
XM$V_MDM_MAINT1	Maintenance mode 1 for local loopback
XM$V_MDM_FREQ	Select frequency
XM$V_MDM_RDY	Data terminal ready to receive or transmit data
XM$V_MDM_POLL	Select polling modem mode

2.4.5.2 Read Line Unit Modem Status

The read line unit modem status function reads the DMP11's line unit modem register. The VAX/VMS operating system provides the following combination of function code and modifier:

- IO$_SENSEMODE!IO$M_RD_MODEM—read line unit modem status
- IO$_SENSEMODE!IO$M_CTRL!IO$M_RD_MODEM—read line unit modem status (DMF32)

These codes take the following device/function-dependent argument:

- P1—the address of a longword buffer into which the line unit's modem status is stored. One or more of the bits listed in the following table can be set in the buffer.

Bit	Meaning
XM$V_MDM_CARRDET[1]	Receiver is active (Carrier Detect)
XM$V_MDM_MSTNDBY	STANDBY indication from modem
XM$V_MDM_CTS[1]	Data can be transmitted (CTS)
XM$V_MDM_DSR[1]	Modem is in service (DSR)
XM$V_MDM_HDX	Line unit is set to half-duplex mode
XM$V_MDM_RTS[1]	Request to send data from USART (RTS)
XM$V_MDM_DTR[1]	Line unit is available and online (DTR)
XM$V_MDM_RING[1]	Modem has just been dialed up (RING)
XM$V_MDM_MODTEST	Modem is in TEST MODE
XM$V_MDM_SIGQUAL	SIGNAL QUALITY from modem interface
XM$V_MDM_SIGRATE	SIGNAL RATE from modem interface

[1]Only for the DMF32

2.4.5.3 Read Device Status Slot

The read device status slot function reads a particular one-word memory location in a global or specified tributary status slot in the DMP11 controller. It is not supported for the DMF32. The VAX/VMS operating system provides two combinations of function code and modifier:

- IO$_SENSEMODE!IO$M_RD_MEM!IO$M_CTRL—read global status slot
- IO$_SENSEMODE!IO$M_RD_MEM—read tributary status slot

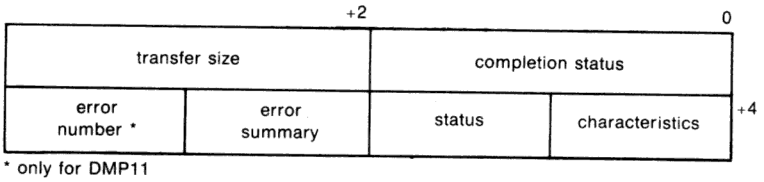
These codes take the following device/function-dependent arguments:

- P1—the address of a longword buffer where the status slot information is stored
- P2—the tributary status slot address (0–31)

2.5 I/O Status Block

The I/O status block (IOSB) for all DMP11, DMF32, and asynchronous DDCMP functions is shown in Figure 2–5. Appendix A lists the completion status returns for these functions. (The *VAX/VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.)

Figure 2–5 IOSB Contents



The first longword of the IOSB returns, in addition to the completion status, either the size (in bytes) of the data transfer or the size (in bytes) of the extended characteristics buffer returned by a sense mode function. The second longword returns the unit characteristics listed in Table 2–2, the line status bits listed in Table 2–3, the error summary bits listed in Table 2–4, and for the DMP11, the total number of errors accrued.

2.6 Programming Example

This sample program (Example 2–1) shows the typical use of QIO functions in DMP11 and DMF32 driver operations such as starting the controller and tributary, and transmitting and receiving data.

To run the following program on the first DMP11 in the system, enter the initial DCL command.

Example 2–1 DMP11/DMF32 Program Example

```
        $ ASSIGN XDAO: DEV
        .TITLE  EXAMPLE - DMP11/DMF32  Device Driver Sample Program
        .IDENT  'X00'
        $IODEF                      ; Define I/O functions and modes
        $NMADEF                     ; Define Network Management symbols
        $XMDEF                      ; Define driver status flags

;
;  Macro definitions
;
        .macro  type      string,?l             ;
        store   <string>                        ;
        movl    #$$.tmpx,cmdorab+rab$l_rbf      ;
        movw    #$$.tmpx1,cmdorab+rab$w_rsz     ;
        $put    rab=cmdorab                     ;
        blbs    r0,1                            ;
        $exit_s                                 ;
1:                                              ;
        .endm   type                            ;
;
        .macro  store     string,pre
        .save
        .psect  $$$dev
        $$.tmpx=.
        pre
        .ascii  %string%
        $$.tmpx1=.-$$.tmpx
        .restore
        .endm   store
CMDOFAB:        $FAB    fac=put,fnm=sys$output:,- ; Output FAB
                        mrs=132,rat=cr,rfm=var
CMDORAB:        $RAB    ubf=cmdbuf,usz=cmdbsz,- ; Output RAB
                        fab=cmdofab
CMDBUF::        .BLKB   256                     ; Command buffer
CMDBSZ=         .-CMDBUF                        ; Buffer size
FAOBUFDSC:      .LONG   CMDBSZ,CMDBUF           ; FAO buffer
                                                ; descriptor
FAOLEN:         .BLKL   1                       ; FAO output buffer
                                                ; length
P2BUF::         .BLKL   50                      ; P2 buffer
P2BUFSZ=        .-P2BUF                         ; P2 buffer size
P2BUFDSC:       .LONG   P2BUFSZ,P2BUF           ; P2 buffer descriptor
P1BUF::         .BLKQ   1                       ; P1 buffer
P1BUFSZ=        .-P1BUF                         ; P1 buffer size
CHNL::          .BLKL   1                       ; Channel number
IOSB::          .BLKL   1                       ; I/O status block
DEVDSC:         .ASCID 'DEV'                    ; Device to assign
QIOREQDSC:      .LONG   QIOREQSZ,QIOREQ         ; QIO request status
QIOREQ:         .ASCII  'QIO completion status = !XL'
                .ASCII  'IOSB1 = !XL, IOSB2 = !XL'
QIOREQSZ=       .-QIOREQ                        ; Size of QIO status
                                                ; report
XMTBUFLEN=512                                   ; Size of transmit
                                                ; buffer
XMTBUF:         .REPEAT XMTBUFLEN
                .BYTE   ^X93                    ; Transmit data
                .ENDR
RCVBUF:         .BLKB   XMTBUFLEN
```

(Continued on next page)

Example 2–1 (Cont.) DMP11/DMF32 Program Example

```
;
;  This is the start of the program section
;
START:: .WORD     0
        $OPEN     FAB=CMDOFAB                   ; Open output
        BLBC      R0,EXIT                       ;
        $CONNECT  RAB=CMDORAB                   ; Connect to output
        BLBC      R0,EXIT                       ;
        BRB       CONT                          ; Continue
EXIT:   $EXIT_S                                 ; Exit program
CONT:   TYPE      <DMP11/DMF32 Test Program>
        TYPE      < >
        $ASSIGN_S DEVNAM=DEVDSC,CHAN=CHNL       ; Assign unit
        BLBC      R0,EXIT                       ; Exit on error
;
;  Initialize and start controller
;
        MOVZWL    #XM$M_CHR_LOOPB!XM$M_CHR_DMC,P1BUF+4 ; Set P1 flags,
                                                ; loopback and DMC
                                                ; compatible
        MOVW      #XMTBUFLEN,P1BUF+2            ; Set P1 buffer size
        CLRL      P2BUFDSC                      ; Set zero length P2
                                                ; buffer
        BSBW      INIT                          ; Issue QIO
;
;  Establish and start tributary
;
        CLRQ      P1BUF                         ; Reset P1 buffer
        MOVAB     P2BUF,R7                      ; Get address of P2
                                                ; buffer
        MOVW      #NMA$C_PCCI_TRI,(R7)+         ; Set parameter code
        MOVZBL    #1,(R7)+                      ; Store trib address

        MOVZBL    #6,P2BUFDSC                   ; Store length of P2
                                                ; buffer
        BSBW      ESTAB                         ; Issue QIO
;
;  Loopback data
;
        MOVZWL    #100,R9                       ; Loop device 100
                                                ; times
10$:    BSBW      XMIT                          ; Issue transmit
        BSBW      RECV                          ; Issue receive
        MOVAB     XMTBUF,R1                     ; Get address of
                                                ; transmit data
        MOVAB     RCVBUF,R2                     ; Get address of
                                                ; received data
        MOVZWL    #XMTBUFLEN,R3                 ; Get number of bytes
                                                ; to verify
20$:    CMPB      (R1)+,(R2)+                   ; Check data
        BNEQ      30$                           ;
        SOBGTR    R3,20$                        ;
        SOBGTR    R9,10$                        ;
        BRW       EXIT                          ; Exit
30$     TYPE      <*** Loopback buffer comparison error ***>
        BRW       EXIT                          ; Exit
```

(Continued on next page)

Example 2–1 (Cont.) DMP11/DMF32 Program Example

```
;
;  Initialize controller QIO
;
INIT:   TYPE    <*** Initialize controller QIO ***>
        $QIO_S  chan=chnl,func=#io$_setchar!io$m_ctrl!io$m_startup,-
                p1=p1buf,p2=#p2bufdsc,iosb=iosb,p3=#5
        BRW     QIO_STATUS                      ;
;
;  Start tributary QIO
;
ESTAB:  TYPE    <*** Startup tributary QIO ***>
        $QIO_S  chan=chnl,func=#io$_setchar!io$m_startup,-
                p1=p1buf,p2=#p2bufdsc,iosb=iosb
        BRW     QIO_STATUS
;
;  Transmit data QIO
;
XMIT:   TYPE    <*** Transmit buffer QIO ***>
        $QIO_S  chan=chnl,func=#io$_writevblk,p1=xmtbuf,-
                p2=#xmtbuflen,iosb=iosb
        BRW     QIO_XMTST
;
;  Receive data QIO
;
RECV:   TYPE    <*** Receive buffer QIO ***>
        $QIO_S  chan=chnl,efn=#2,func=#io$_readvblk,p1=rcvbuf,-
                p2=#xmtbuflen,iosb=iosb
        .BRB    qio_status
        .ENABL  LSB
QIO_STATUS:                                     ; Check status of QIO
        BLBC    IOSB,10$                        ; Br if error on QIO
QIO_XMTST:                                      ; Check status of XMIT
        BLBC    R0,10$                          ; Br if error on
        RSB                                     ; request, else return
                                                ; to caller
10$     MOVZWL  IOSB,R1                         ; Get I/O status block
        PUSHL   R1                              ; Push I/O status block
        PUSHL   R0                              ; Push system service
                                                ; status
        PUSHAQ  FAOBUFDSC                       ; Push address of FAO
                                                ; buffer descriptor
        PUSHAW  FAOLEN                          ; Push address of
                                                ; output length
        PUSHAQ  QIOREQDSC                       ; Push address of
                                                ; input string
        CALLS   #5,G^SYS$FAO                    ; Get error message
        MOVAB   CMDBUF,CMDORAB+RAB$L_RBF        ; Get output buffer
                                                ; address
        MOVW    FAOLEN,CMDORAB+RAB$W_RSZ        ; Get output buffer
                                                ; length
        $PUT    CMDORAB                         ; Print error test
        BRW     EXIT                            ; Exit
        .DSABL  LSB
        .END    START
```

3 DR11–W and DRV11-WA Interface Driver

This section describes the use of the DR11–W interface driver (XADRIVER). (The DRV11-WA uses the same driver; unless otherwise stated, references to the DR11–W also apply to the DRV11-WA.)

3.1 Supported Devices

The DR11–W is a general-purpose, 16-bit, parallel direct-memory-access (DMA) data interface. It can be used either as an interface between memory and a user device or as an interprocessor link (non-DECnet) between two systems.

The DRV11-WA is similar to the DR11–W. However, it operates as an interface device that uses the 22-bit Q-bus, rather than the UNIBUS. Unless otherwise indicated, the DRV11-WA driver performs the same QIO functions as the DR11–W driver; descriptions of DR11–W features also apply to the DRV11-WA. The DRV11-WA driver is supported for the MicroVAX II, but not the MicroVAX I.

Note: The XADRIVER documentation and the XADRIVER source code in Version 4.4 of the VAX/VMS operating system assume that the DRV11-WA is at CS Revision Level B and Etch Revision Level D or earlier. If subsequent revisions are made to the board, the customer may need to modify the driver source code, which is available in SYS$EXAMPLES:XADRIVER.MAR, to accommodate changes in the hardware. No such restrictions apply to the DR11–W.

A DR11–W may be linked to another DR11–W and a DR11–W may be linked to a DRV11-WA, but it is not possible to link two DRV11-WAs together.

Figure 3–1 shows two typical applications of the DR11–W and DRV11-WA.

The driver (XADRIVER) allows general access to the features provided by the DR11–W and DRV11-WA devices. Function codes and modifiers are provided to control, and to transfer data between, the user device and the VAX/VMS operating system.

3.1.1 Device Differences

The DR11–W and the DRV11-WA do not work exactly the same way. Differences that affect the user at the QIO interface level are listed below; the referenced sections contain additional information on these differences.

- Unsolicited interrupts—The DRV11-WA driver does not acknowledge unsolicited interrupts (see Section 3.3).
- IO$M_WORD function modifier—The DRV11-WA driver does not perform word mode transfers (see Section 3.3).
- CSR error bit—The DRV11-WA driver detects some of, but not all, the hardware errors detected by the DR11–W driver (see Section 3.1.6).

- Error information register (EIR)—The DRV11-WA does not have an EIR (see Section 3.1.6).
- IO$M_RESET function modifier—The DRV11-WA cannot be reset in the same way as the DR11–W. (see Section 3.3).
- IO$M_DATAPATH function modifier—The IO$M_DATAPATH function modifier is ignored for the DRV11-WA driver (see Section 3.3.3.1).

3.1.2 DRV11-WA Installation

In addition to the two installation considerations described in this section, the user should follow the instructions in the hardware documentation when installing the DRV11-WA.

3.1.2.1 Type of Addressing

Bit 10 of the vector address selection switch is not used as part of the vector, but rather selects 18- or 22-bit addressing. Set the device to 22-bit addressing.

3.1.2.2 Device Address and Interrupt Vector Address Selection

Because the DRV11-WA is designed to be compatible with the DR11-B, the hardware documentation instructs the user to set the device address and the interrupt vector address to those reserved for the DR11-B. However, under MicroVMS, the DRV11-WA is treated as much as possible like a DR11–W. Therefore, the user must set the device address and interrupt vector address to those reserved for the DR11–W for the device to autoconfigure correctly; namely, set the device address to rank 19 and the interrupt vector address to rank 40, both in floating address space. The exact addresses can be calculated by hand or with the help of the VAX/VMS System Generation Utility CONFIGURE command.

If the user desires to set up the device at the DR11-B address as described in the hardware documentation, the device should be configured using the following commands:

```
$run sys$system:sysgen
load sys$system:xadriver
connect xaa0 /adap=ub0/csr=%o772410/vector=%o124
exit
```

3.1.3 DR11–W and DRV11-WA Transfer Modes

The DR11–W transfers data in two ways: in block mode and in word mode. (Word mode transfers are not possible with the DRV11-WA.) In block mode, all transfers are provided by the DMA facility. Each QIO request moves a single buffer of data between the user device and physical memory. One interrupt is generated on completion of the transfer. The transfer rate and transfer direction are controlled by the user device.

In block mode two types of UNIBUS or Q-bus transfers are possible: single cycle and burst. During single-cycle transfers the bus is arbitrated for each word (two bytes) of information exchanged. Both the DR11–W and the DRV11-WA have a single cycle mode that is supported by VAX/VMS and MicroVMS.

Burst transfers result in the exchange of multiple words without arbitration of the bus. Two classes of burst mode transfers are possible, depending on the position of a switch on the module. On the DR11–W, the VAX/VMS operating system only permits the use of dual cycle mode (class 1) in which two words are transferred for each arbitration of the UNIBUS. On the DRV11-WA, MicroVMS only permits the use of the 4-cycle mode in which four words are transferred for each arbitration of the Q-bus. Burst mode transfers must be used with caution. They can provide greater performance but can prevent use of the bus by other devices for what might be unacceptable periods. Both the DR11–W and the DRV11-WA also have an N-cycle burst mode that cannot be used on VAX/VMS or MicroVMS systems. On DRV11-WA boards prior to CS Revision Level B and Etch Revision Level D, N-cycle is the only form of burst mode available, and there is no burst mode selection switch on the module.

In word mode a single QIO request transfers a buffer of data, with an interrupt requested for each word. Word mode is usually used to exchange control information between the application program and the user device. Once the proper control information has been accepted, a block-mode transfer can be started to exchange data.

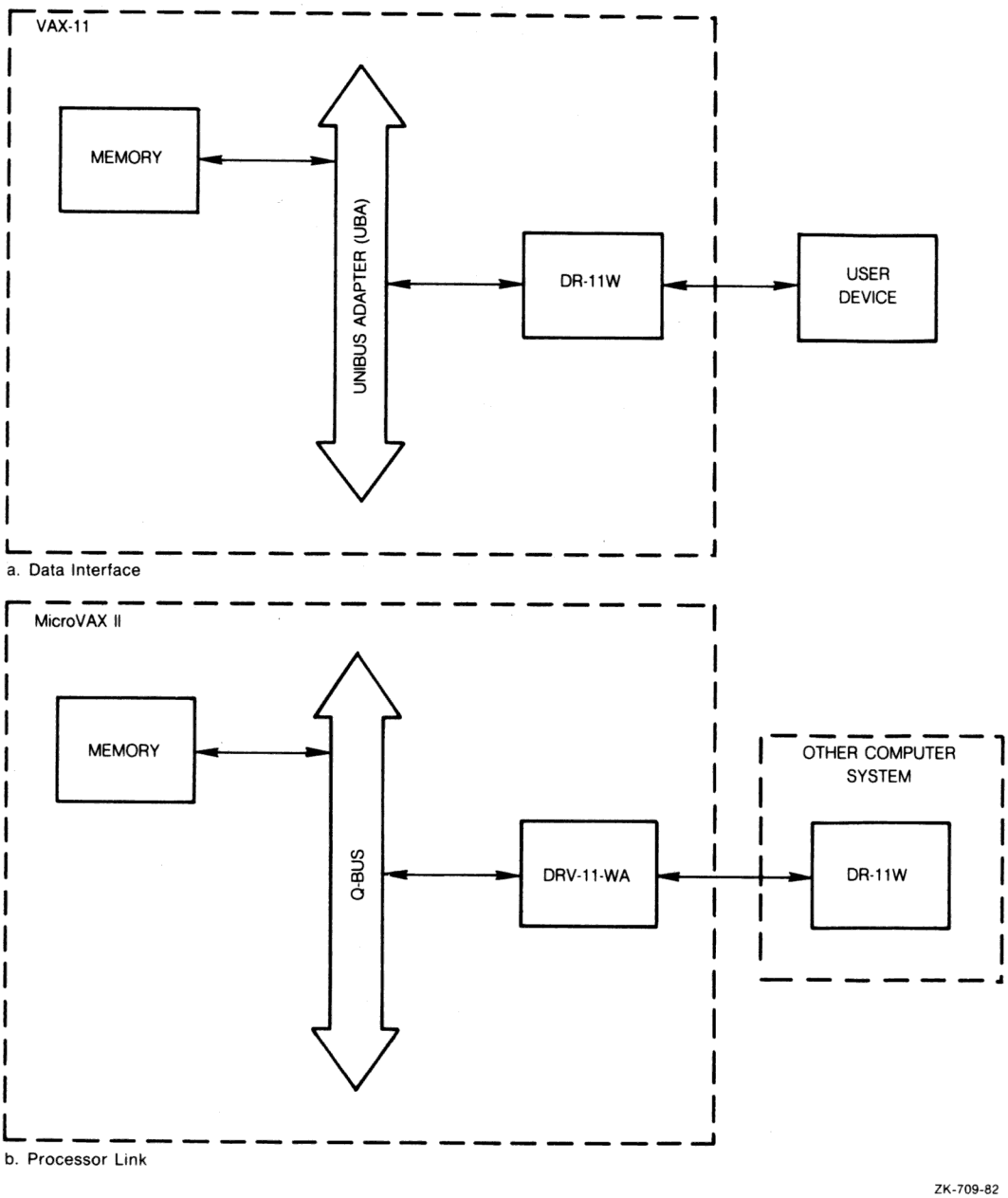
In both block- and word-mode transfers, the transfer size is indicated by the byte count value specified in the P2 argument. The DR11–W and DRV11-WA transfer information between main memory and the user device in one-word (two-byte) units; transfers are counted on a word-by-word basis. However, the VAX/VMS operating system counts information one byte at a time. Consequently, if the desired DR11–W or DRV11-WA transfer is 100 words, the P2 argument must specify 200 (bytes) for the transfer count value. If an odd number of bytes is specified for the transfer count, the driver will reject the QIO request.

Transfers to and from memory typically occur from sequentially increasing addresses. The user device can inhibit the increment to the next address.

During block mode transfers, the user device controls the transfer direction through signals exchanged with the driver. Neither the VAX/VMS operating system nor the application program has any control over the transfer direction. Consequently, a read or write request to the driver by the application program should be by convention, according to the intended action. An effect of this, regardless of whether a read or write QIO function is specified, is that the application program's data buffer is always checked for modify access (rather than read or write access) during block-mode transfers. In word mode, the transfer direction is controlled explicitly by the device driver.

The terms read and write must be used with caution when discussing transfers. This manual uses these terms for the application procedure running under the VAX/VMS operating system. In other words, a read operation involves the transfer of information from the user device to VAX memory. A write operation involves the transfer of information from VAX memory to the user device. Receive and input are synonymous with read operations; transmit and output are synonymous with write operations.

Figure 3–1 Typical DR11–W/DRV11-WA Device Configurations



3.1.4 DR11–W and DRV11-WA Control and Status Register Functions

For each buffer of data transferred, the DR11–W or DRV11-WA driver allows for the exchange of an additional six bits of information: the function (FNCT) and status (STATUS) bits, which are included in the control and status register (CSR). These bits are accessible to an application process through the device driver QIO interface. The FNCT bits are labeled FNCT 1, FNCT 2, and FNCT 3. The STATUS bits are labeled STATUS A, STATUS B, and STATUS C.

The user device interfaced to the DR11–W or DRV11-WA interprets the value of the three FNCT bits. The QIO request that initiates the transfer specifies the IO$M_SETFNCT modifier to indicate a change in the value for the FNCT bits. The P4 argument of the request specifies this value. P4 bits 0 through 2 correspond to FNCT bits 1,2,3, respectively. Bits 3 through 31 are not used. If required, the FNCT bits must be set for each request. The FNCT bits set in the CSR are passed directly to the user device.

The DR11–W and DRV11-WA STATUS bits are available in bits 9 through 11 of the CSR, which correspond to STATUS bits C,B,A, respectively. On completion of all transfers, the STATUS bits are returned from the user device through the DR11–W or DRV11-WA to the IOSB. Neither the VAX/VMS operating system nor the DR11–W/ DRV11-WA modifies these bits in any way. Thus, both FNCT and STATUS fields are defined solely by the user device. Except when used as an interprocessor link, the DR11–W or DRV11-WA takes no special action based on the state of these fields, and the FNCT bits remain set until explicitly changed with the IO$M_ SETFNCT function modifier.

The DR11–W and DRV11-WA CSR STATUS bits should not be confused with the status values returned in the I/O status block.

The function modifier IO$M_CYCLE sets the CSR CYCLE bit for the transfer specified by the QIO request. In block mode, the CYCLE bit initiates the transfer of the first word of data. In word mode, IO$M_CYCLE has no effect.

Section 3.1.7 describes the special meaning given to the FNCT and STATUS bits by the DR11–W or DRV11-WA hardware and device driver when used as an interprocessor link.

3.1.5 Data Registers

Two registers are used to transfer information to and from the user device. The input data register (IDR) contains the last data value transferred into the DR11–W or DRV11-WA from the user device. The output data register (ODR) contains the last value transferred from the DR11–W or DRV11-WA to the user device. During block-mode operations, these registers are controlled automatically and require no explicit action on the part of the application program. During word-mode write operations, the DR11–W driver loads the ODR with each successive data word; each word is then available to the user device. During word-mode read operations, the driver reads the IDR and stores the value in memory. Interrupts from the DR11–W synchronize reading and writing the IDR and ODR when in word mode.

3.1.6 Error Reporting

The error information register (EIR) is used for reporting certain types of error conditions to the application program at the completion of each request. As the result of a user device action, the device sets the ATTN bit in the CSR. The CSR ERROR bit is also set at this time. If ERROR is set during a block-mode transfer, the transfer is aborted. Table 3–5 in Section 3.4 lists the EIR and CSR bit assignments for the I/O status block.

The DRV11-WA detects some, but not all, types of errors detected by the DR11–W. Specifically, the error bit in the CSR (bit 15) for the DRV11-WA signals attention interrupts, nonexistent memory errors, and power failures at the remote device, but does not signal multicycle request errors or parity errors. The DRV11-WA does not have an EIR. The driver always returns

zeros in place of the EIR in the fourth word of the IOSB when an I/O operation is completed.

3.1.7 Link Mode of Operation

The XADRIVER driver can control two DR11–Ws, or a DR11–W and a DRV11-WA, connected as an interprocessor link between two computer systems. Control switches on the DR11–W module are set to place the hardware in this configuration. No such control switches are available or necessary on the DRV11-WA. Either the set mode or the set characteristics function must also be used to instruct the driver to function in link mode.

The DRV11-WA cannot be linked with another DRV11-WA due to a hardware restriction in the device.

In link operations two cooperating processes exchange data through the devices, which function as a memory-to-memory interface. This feature requires that the two processes agree on, and establish a basis for describing, the direction of the data transfer, the message sizes, and for arbitrating use of the link.

In link operations, the FNCT and STATUS bits are given special meaning by the DR11–W or DRV11-WA hardware and the device driver. Proper operation of the DR11–W or DRV11-WA as an interprocessor link depends on the correct use of these bits. The driver does not enforce correct use of the FNCT and STATUS bits. When issuing a QIO request to the DR11–W or DRV11-WA in link mode with IO$M_SETFNCT specified, the correct values and sequence of FNCT bits must be provided by the application image. Table 3–1 lists the FNCT and STATUS bits and what actions occur when the DR11–W or DRV11-WA is in link mode. (Table 3–5 lists the CSR bit assignments.)

Table 3–1 Control and Status Register FNCT and STATUS Bits (Link Mode)

Bit	Function
FNCT 1	Indicates whether the DR11–W or DRV11-WA at this end of the link is to transmit or receive data. If FNCT 1 is 0, the DR11–W or DRV11-WA transmits data from memory to the associated DR11–W or DRV11-WA at the other end of the link. If FNCT 1 is 1, the DR11–W or DRV11-WA receives data from the associated DR11–W or DRV11-WA and stores it in memory. (Note that two DRV11-WAs cannot be linked together.) For proper operation, one system must set FNCT 1 to 1 (for receive) and the associated system must set FNCT 1 to 0 (for transmit).
FNCT 2	Interrupts the remote processor. For proper operation, the driver must be set to operate as a link. When a set mode or set characteristics function is used to instruct the driver to perform a link operation, the driver does not leave FNCT 2 set. Instead, the driver sets and then immediately clears the bit to provide a pulse, rather than a level, to the associated system.

Table 3–1 (Cont.) Control and Status Register FNCT and STATUS Bits (Link Mode)

Bit	Function
FNCT 3	Indicates whether the nonprocessor request (NPR) transfers that follow occur as single-cycle or burst-mode transfers. If FNCT 3 is 0, burst transfers are performed. If FNCT 3 is 1, single-cycle transfers are performed. Users should note that burst-mode transfers can occupy the UNIBUS or Q-bus for long periods, to the exclusion of other devices on the same bus.
STATUS A	Returns the value of FNCT 3 set in the associated computer system. When an interrupt is returned from the associated computer denoting the need to exchange a message, STATUS A indicates whether the request that follows is to be set up for single-cycle or for burst operation.
STATUS B	Returns the value of FNCT 2 set in the associated system. Because the DR11–W driver, when configured as a link, never leaves FNCT 2 set, STATUS B is never read as a 1. When STATUS B is set, ATTENTION and, in turn ERROR, are set in the DR11–W or DRV11-WA. When the driver handles the resulting interrupt, it attempts to clear ATTENTION. If ATTENTION cannot be cleared, it indicates that the condition causing it was a level, held true by the associated system. Since ATTENTION can be set by conditions other than FNCT 2, for example, the error ACLO in the associated system, treating FNCT 2 as a pulse allows the receiving DR11–W to differentiate between an error and a normal processor interrupt request.
STATUS C	Returns the value of the FNCT 1 bit sent by the associated computer. STATUS C indicates whether the DMA transfer that follows is a transmit or a receive operation.

If a DR11–W in link configuration sets one or more of the three CSR FNCT bits, the other DR11–W will perform one or more of the following actions:

- Request an interrupt
- Specify the intended transfer direction for a block-mode transfer that follows
- Declare whether the transfer is to take place in burst or single-cycle operation

In each case, the value written into the FNCT bits of the first DR11–W is available and is read from the STATUS bits of the other DR11–W.

Since either process can initiate the data transfer, arbitration for the use of the link is automatic. If both processes want to write or both want to read, a timeout occurs. A timeout also occurs if either process neglects to specify the agreed-upon transfer direction or message size. Each process should specify a different timeout period or a different time before re-requesting the link after a timeout. These actions, which preclude a lockup of the link, are not enforced by the driver.

If an attention interrupt is generated, it indicates that either the DR11–W or DRV11-WA associated with the other system is initiating a transfer, or that the other DR11–W or DRV11-WA is going off line because of a power failure. The DR11–W driver ability to clear ATTENTION (see description of STATUS

B in Table 3–1) allows a data transfer to be distinguished from a hardware error. If an error occurs and ATTENTION can be cleared, SS$_DRVERR is returned as the status. If ATTENTION cannot be cleared, SS$_CTRLERR is returned.

3.2 Device Information

Users can obtain information on DR11–W or DRV11-WA characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume*.)

$GETDVI returns DR11–W- or DRV11-WA-specific characteristics when you specify the item codes DVI$_DEVCHAR and DVI$_DEVDEPEND. Tables 3–2 and 3–3 list these characteristics. The $DEVDEF macro defines the device-independent characteristics; the $XADEF macro defines the device-dependent characteristics.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and device class names, which are defined by the $DCDEF macro. The device type for the DR11–W is DT$_DR11W; the device type for the DRV11-WA is DT$_XA_DRV11WA. The device class for both the DR11–W and DRV11-WA is DC$_REALTIME. DVI$_DEVBUFSIZ returns the default buffer size, which is 65,535.

Table 3–2 DR11–W and DRV11-WA Device-Independent Characteristics

Characteristic[1]	Meaning
Dynamic Bits (Conditionally Set)	
DEV$M_AVL	Device is online and available
DEV$M_ELG	Error logging is enabled for this device.
Static Bits (Always Set)	
DEV$M_IDV	Input device
DEV$M_ODV	Output device
DEV$M_RTM	Real-time device

[1]Defined by the $DEVDEF macro.

Table 3–3 DR11–W and DRV11-WA Device-Dependent Characteristics

Value[1]	Meaning
XA$M_DATAPATH	Describes which UNIBUS adapter data path is in use. 0 = direct data path; 1 = buffered data path. The initial state of this bit is 0. (Not applicable to the DRV11-WA.)
XA$M_LINK	Describes whether the DR11–W or DRV11-WA is used as a link or as a user device interface. 0 = user device interface; 1 = link. The initial state of this bit is 0.

[1]Defined by the $XADEF macro.

3.3 DR11–W and DRV11-WA Function Codes

The XADRIVER can perform logical, virtual, and physical I/O operations. The basic I/O functions are read, write, set mode, and set characteristics. Table 3–4 lists these functions and their function codes. The following paragraphs describe these functions in greater detail.

Table 3–4 DR11–W/DRV11-WA I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_READLBLK P1,P2,- P3,P4,P5	L	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Read logical block.
IO$_READVBLK P1,P2,- P3,P4,P5	V	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Read virtual block.
IO$_READPBLK P1,P2,- P3,P4,P5	P	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Read physical block.
IO$_WRITELBLK P1,P2,- P3,P4,P5	L	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Write logical block.
IO$_WRITEVBLK P1,P2,- P3,P4,P5	V	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Write virtual block.
IO$_WRITEPBLK P1,P2,- P3,P4,P5	P	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Write physical block.
IO$_SETMODE P1,P3	L	IO$M_ATTNAST	Set DR11–W or DRV11-WA characteristics for subsequent operations.

[1] V = virtual, L = logical, P = physical (There is no functional difference in these operations.)

[2] Not applicable to the DRV11-WA

Table 3–4 (Cont.) DR11–W/DRV11-WA I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_SETCHAR P1,P3	P	IO$M_ATTNAST IO$M_DATAPATH	Set DR11–W or DRV11-WA characteristics for subsequent operations.

[1]V = virtual, L = logical, P = physical (There is no functional difference in these operations.)

Although the XADRIVER does not differentiate among logical, virtual, and physical I/O functions (all are treated identically), the user must have the required privilege to issue a request.

The read and write functions take the following device/function-dependent arguments:

- P1—the starting virtual address of the buffer that is to receive data for a read operation; or the virtual address of the buffer that is to send data to the DR11–W for a write operation. Modify access to the buffer, rather than read or write access, is checked for all block-mode read and write requests.
- P2—the size of the data buffer in bytes, that is, the transfer count. Since the DR11–W performs word transfers, the transfer count must be an even value. The maximum transfer size is 65,534 bytes. If a larger number is specified, the high-order bits of this field are ignored.
- P3—the timeout period for this request (in seconds). The value specified must be equal to or greater than 2. IO$M_TIMED must be specified. The default timeout value for each request is 10 seconds.
- P4—the value of the DR11–W command and status register (CSR) function (FNCT) bits to be set. If IO$M_SETFNCT is specified, the low-order three bits of P4 (2:0) are written to the CSR FNCT bits 3:1 (respectively) at the time of the transfer.
- P5—the value (low two bytes) to be loaded into the DR11–W output data register (ODR). IO$M_SETFNCT must be specified and the transfer count (P2) must be 0.

If a direct data path (DDP) is used (see Section 3.3.3.1), the address specified by the P1 argument must be word-aligned. However, if a buffered data path (BDP) is used, byte alignment is allowed. All transfers through the BDP, which is only available on the UNIBUS, must occur from sequential, increasing addresses. If the user device interfaced to the DR11–W cannot conform to this requirement, the DDP must be used.

The transfer count specified by the P2 argument must be an even number of bytes. If an odd number is specified, an error (SS$_BADPARAM) is returned in the I/O status block (IOSB). If the transfer count is 0, the driver will transfer no data. However, if IO$M_SETFNCT is specified and P2 is 0, the driver will set the FNCT bits in the DR11–W CSR, load the low two bytes specified in P5 into the DR11–W ODR, and return the current CSR status bit values in the IOSB.

The read and write functions can take five function modifiers:

- IO$M_SETFNCT—sets the FNCT bits in the DR11–W CSR before the data transfer is initiated. The low-order three bits of the P4 argument specify the FNCT bits. The user device that interfaces with the DR11–W or DRV11-WA receives the FNCT bits directly, and their value is interpreted entirely by the device.

  Additionally, if the transfer count (P2) is 0, load the value specified in P5 into the device ODR.

  If a link operation is specified in the device-dependent characteristics (XA$M_LINK = 1), FNCT 2 will not be left set (that is, it will be set and immediately cleared) in the device CSR.

- IO$M_WORD—performs the data transfer in word mode rather than in DMA block mode (not applicable to the DRV11-WA). In word mode an interrupt occurs for each word transferred. This allows the exchange of a small amount of data to establish the parameters for a block-mode data transfer that follows.

  If IO$M_WORD is included in a write request, the first word in a user's buffer is loaded into the DR11–W ODR. The driver then waits for an interrupt before proceeding to load the next word or complete the request. If IO$M_WORD is included in a read request, the driver waits for an interrupt and then reads a word from the DR11–W IDR and stores it in the user's buffer.

  Interrupts are initiated when either the user device or, when in link operation, the associated DR11–W sets ATTENTION.

  If the DR11–W or DRV11-WA receives an unsolicited interrupt, no read or write request is posted. If the next request is for a word-mode read, the driver will return the word read from the DR11–W IDR and store it in the first word of the user's buffer. In this case the driver does not wait for an interrupt.

  The DRV11-WA does not respond to unsolicited interrupts from a remote device; the DRV11-WA only acknowledges interrupts when a DMA transfer is outstanding. Consequently, word-mode transfers are not possible on a DRV11-WA because the device does not acknowledge the interrupt that occurs when the I/O operation is completed; the QIO waits indefinitely or times out. (In some cases, the user can work around this problem by causing the remote device to generate an interrupt, which makes the local DRV11-WA complete the I/O operation with an SS$_OPINCOMPL status.)

- IO$M_TIMED—uses the timeout value in the P3 argument rather than the default timeout value of 10 seconds.

- IO$M_CYCLE—sets the cycle bit in the DR11–W or DRV11-WA CSR for this request. In block mode, this initiates the first NPR cycle. For user devices, the application of the cycle bit is dependent on the specific device. In word mode, IO$M_CYCLE is ignored. In link operations, only the transmitting DR11–W or DRV11-WA must set CYCLE and then only after the companion DR11–W has its receive request initiated.

- IO$M_RESET—performs a device reset to the DR11–W before any I/O operation is initiated. This function does not affect any other device on the system.

The DRV11-WA can be reset only by initializing the Q-bus and all other devices attached to the Q-bus. Therefore, when the IO$M_RESET function modifier is used to reset the DRV11-WA, the XADRIVER simulates a reset by setting the word count register (WCR) to indicate one word left to be transferred and setting the CYCLE bit to complete the transfer. If the driver is not performing a transfer at the time of a reset, the reset is a NOOP.

On completion of each read or write request, including those requests with a zero transfer count, the current value of the DR11–W or DRV11-WA CSR and DR11–W EIR is returned in the I/O status block.

3.3.1 Read

Read functions provide for the direct transfer of data from the user device that interfaces with the DR11–W or DRV11-WA into the user process's virtual memory address space. The VAX/VMS operating system provides three function codes:

- IO$_READLBLK—read logical block
- IO$_READVBLK—read virtual block
- IO$_READPBLK—read physical block

Five function-dependent arguments and five function modifiers are used with these codes. These arguments and modifiers are described at the beginning of Section 3.3.

3.3.2 Write

Write functions provide for the direct transfer of data to the user device that interfaces with the DR11–W or DRV11-WA from the user process's virtual memory address space. The VAX/VMS operating system provides three function codes:

- IO$_WRITELBLK—write logical block
- IO$_WRITEVBLK—write virtual block
- IO$_WRITEPBLK—write physical block

Five function-dependent arguments and five function modifiers are used with these codes. These arguments and modifiers are described at the beginning of Section 3.3.

3.3.3 Set Mode and Set Characteristics

Set mode operations affect the operation and characteristics of the associated DR11–W or DRV11-WA. The VAX/VMS operating system defines two types of set mode functions: set mode and set characteristics. Set mode requires logical I/O privilege. Set characteristics requires physical I/O

privilege. These functions allow the user process to set or change the device characteristics. Two function codes are provided:

- IO$_SETMODE—set mode
- IO$_SETCHAR—set characteristics

These functions take the following device/function-dependent arguments:

- P1—the virtual address of a quadword characteristics buffer. If the function modifier IO$M_ATTNAST is specified, P1 is the address of the AST service routine. In this case, if P1 is 0, all attention ASTs are disabled.
- P3—the access mode to deliver the AST (maximized with the requester's access mode). If IO$M_ATTNAST is not specified, P3 is ignored.

Figure 3–2 shows the quadword P1 characteristics buffer for IO$_SETMODE and IO$_SETCHAR.

Figure 3–2 P1 Characteristics Buffer

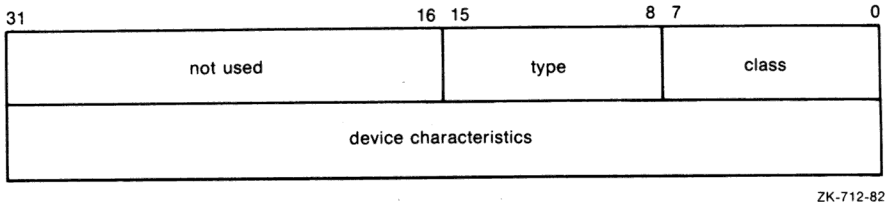


Table 3–3 lists the device characteristics for the set mode and set characteristics functions. The device class value is DC$_REALTIME. The device type value is DT$_DR11W or DT$_XA_DRV11WA. These values are defined by the $DCDEF macro.

3.3.3.1 Set Mode Function Modifiers

The IO$_SETMODE and IO$_SETCHAR function codes can take the following function modifier:

- IO$M_ATTNAST—enable attention AST

This function modifier allows the user process to queue an attention AST for delivery when an asynchronous or unsolicited condition is detected by the DR11–W or DRV11-WA driver. Unlike ASTs for other QIO functions, use of this function modifier does not increment the I/O count for the requesting process or lock pages in memory for I/O buffers. Each AST is charged against the user's AST limit.

Attention ASTs are delivered when any of the following occur:

- Any block- or word-mode data transfer request is completed.
- An unsolicited interrupt from the DR11–W occurs. (The DRV11-WA does not respond to unsolicited interrupts.)
- An attention AST is queued and a previous unsolicited interrupt has not been acknowledged.
- A device timeout occurs.

The Cancel I/O on Channel ($CANCEL) system service is used to flush attention ASTs for a specific channel.

The enable attention AST function modifier enables an attention AST to be delivered to the requesting process once only. After the AST occurs, it must be explicitly reenabled by the function modifier before the AST can occur again. This function modifier does not update the device characteristics.

When the AST is delivered, the AST parameter contains the contents of the DR11–W or DRV11-WA CSR in the low two bytes and the value read from the DR11–W or DRV11-WA IDR in the high two bytes.

In addition to IO$M_ATTNAST, the IO$_SETCHAR function code can take the following function modifier:

- IO$M_DATAPATH—use the data path specified by XA$M_DATAPATH in the P1 characteristics buffer

The IO$M_DATAPATH function modifier allows the user to specify either the direct data path (DDP) or a buffered data path (BDP) for block-mode transfers through the UNIBUS adapter.

The device-specific characteristic XA$M_DATAPATH is used to switch between use of the DDP and the BDP. If XA$M_DATAPATH is set, the BDP is used; if clear, the DDP is used. Regardless of the value of XA$M_DATAPATH, the choice of data path has no effect unless the function modifier IO$M_DATAPATH is also specified, which requires physical I/O privilege.

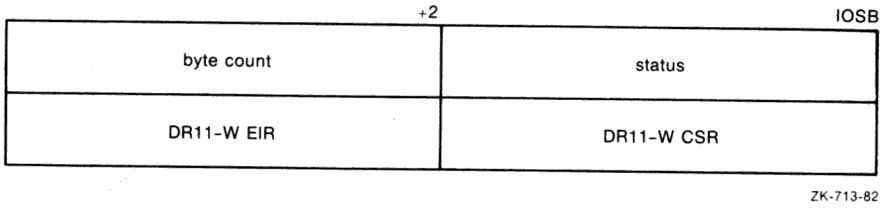
Note: The user should exercise caution when specifying data transfers through the BDP. The user device has access to several hardware functions: C0 and C1, inhibit word count increment, and inhibit bus address increment. If these signals are used out of context of the expected UNIBUS adapter constraints for BDPs, the result is unpredictable.

Unlike the UNIBUS, the Q-bus does not provide a choice between a direct data path and a buffered data path; the IO$M_DATAPATH function modifier is ignored for the DRV11-WA.

3.4 I/O Status Block

The I/O status block (IOSB) for DR11–W or DRV11-WA read and write functions is shown in Figure 3–3. On completion of each read or write request, the I/O status block is filled with system and DR11–W or DRV11-WA status information.

Figure 3–3 IOSB Contents — Read and Write Functions



The first longword of the I/O status block contains I/O status returns and the byte count. Appendix A lists the status returns for read and write functions. (The *VAX/VMS System Messages and Recovery Procedures Reference Manual*

provides explanations and suggested user actions for these returns.) The byte count is the actual number of bytes transferred by the request. If the request ends in an error, the byte count may differ from the requested number of bytes. If a power failure, timeout, or the Cancel I/O on Channel ($CANCEL) system service stops the request, the value in the byte count field is not valid.

The third and fourth words of the I/O status block contain the values of the DR11–W CSR and EIR on completion of the request. (The DRV11-WA has a CSR but not an EIR; the driver always returns zeros in the fourth word of the IOSB when an I/O operation is completed.) Table 3–5 lists the bit assignments for these two words. The *DR11–W User's Manual* provides additional information on the EIR and CSR.

Table 3–5 EIR and CSR Bit Assignments

Bit		Function
EIR	0	Register flag
	1–7	(not applicable)
	8	N-cycle burst
	9	Burst timeout (sets ERROR)
	10	PARITY (sets ERROR)
	11	ACLO (sets ERROR)
	12	Multicycle request (sets ERROR)
	13	ATTENTION (sets ERROR)
	14	Nonexistent memory (sets ERROR)
	15	ERROR (generates interrupt when set)
CSR	0	GO
	1	FNCT 1
	2	FNCT 2
	3	FNCT 3
	4	Extended bus address 16
	5	Extended bus address 17
	6	Interrupt enable
	7	READY
	8	CYCLE
	9	STATUS C
	10	STATUS B
	11	STATUS A
	12	Maintenance mode
	13	ATTENTION (sets ERROR)
	14	Nonexistent memory (sets ERROR)
	15	ERROR (generates interrupt when set)

3.5 Programming Hints

Since user devices of different or unknown capability can be connected to the interface, the interface may be either insufficient for the desired application or significantly inefficient. For this reason, the source code for the DR11–W/DRV11-WA driver (SYS$EXAMPLES:XADRIVER.MAR) is included in both the VAX/VMS and MicroVMS distribution kits as a template for adding driver features or making a particular operation more efficient. The user should note that the driver is not supported if modifications are made to the source program provided. For additional information the reader should refer to the *DR11–W Direct Memory Interface Module User's Guide*, the *DRV11-WA General Purpose DMA Interface User's Guide*, and the *Writing a Device Driver for VAX/VMS*.

3.6 Programming Example

A sample program residing in the SYS$EXAMPLES directory demonstrates how to perform transfers across a DR11–W to DRV11-WA or a DR11–W to DR11–W interprocessor link. The sample program includes the following modules:

- XALINK.MAR—places the device in link mode
- XAMESSAGE.MAR—performs the actual transfer of data
- XATEST.FOR—solicits parameters for the transfer from the user and calls the XALINK.MAR and XAMESSAGE.MAR modules
- XATEST.COM—compiles and links the sample program

Example 3–1, which consists of the module XAMESSAGE.MAR, shows how an actual memory-to-memory link might be implemented using the XADRIVER. All actions are invoked through the $QIO interface by a nonprivileged image.

The program includes the following features:

- Either system can function as the transmitter or the receiver. For any given exchange, one system must be the transmitter and one must be the receiver.
- Either the transmitter or the receiver can call XAMESSAGE first, which is made possible by the driver's ability to keep track of unsolicited attention interrupts. XAMESSAGE uses this feature for the following reasons:
  - To synchronize the DMA exchange
  - To ensure that the receiver issues the block-mode read request first
  - To ensure that the transmitter sets the CYCLE bit to initiate the first NPR transfer
- If either the transmitter or receiver specifies unequal transfer sizes or does not match the transfer direction, either a timeout occurs or one of the procedures returns an error. The caller must resolve these discrepancies.

Table 3–6 lists the main flow of the program. Note that paths for transmit and receive, and for DR11–W and DRV11-WA, are combined in the same module (XAMESSAGE).

The three parts of Table 3–6 describe the operation of XAMESSAGE in three different device configurations:

- a DRV11-WA transmitting a message to a DR11–W
- a DR11–W transmitting a message to a DRV11-WA
- a DR11–W transmitting a message to another DR11–W

The two right-hand columns describe the action taken by each device involved in the transfer. The leftmost column contains the name of the routine in XAMESSAGE that performs the respective action: MAIN refers to the main routine for XAMESSAGE, AST_GO refers to the AST routine by that name, AST_COM refers to the AST routine called AST_ COMPLETION, and ASYNC means that the action occurs asynchronously and is not controlled directly by any code in XAMESSAGE.

Table 3–6 XAMESSAGE Program Flow

DRV11-WA (transmitter) to DR11–W (receiver)

XAMESSAGE	DRV11-WA (Transmitter)		DR11–W (Receiver)	
MAIN	1.	Issue block mode read request.	1.	Enable attention AST.
AST_GO			2.	Execute attention AST as a result of interrupt from transmitter.
AST_GO			3.	Issue block mode read request.
AST_GO	4.	Complete block mode read request prematurely as a result of the interrupt at the beginning of the receiver's read request.		
AST_GO	5.	Issue block mode write request.		
ASYNC	6.	Perform DMA transfer.	6.	Perform DMA transfer.
AST_COM	7.	Execute completion AST, and check for errors.	7.	Execute completion AST, and check for errors.

Table 3–6 (Cont.) XAMESSAGE Program Flow

DR11–W (transmitter) to DRV11-WA (receiver)

XAMESSAGE	DRV11-WA (Receiver)	DR11–W (Transmitter)
MAIN	1. Issue block mode read.	1. Enable attention AST.
AST_GO		2. Execute attention AST as a result of interrupt from receiver.
AST_GO		3. Issue block mode write request.
ASYNC	4. Perform DMA transfer.	4. Perform DMA transfer.
AST_COM	5. Execute completion AST, and check for errors.	5. Execute completion AST, and check for errors.

DR11–W (transmitter) to DR11–W (receiver)

XAMESSAGE	DR11–W (Receiver)	DR11–W (Transmitter)
MAIN	1. Enable attention AST.	1. Enable attention AST.
MAIN		2. Momentarily set the FNCT2 bit via a 0-length transfer to interrupt the receiver.
AST_GO	3. Execute attention AST as a result of interrupt from transmitter.	
AST_GO	4. Issue block mode read request.	
AST_GO		5. Execute attention AST as a result of interrupt from receiver.
AST_GO		6. Issue block mode write request.
ASYNC	7. Perform DMA transfer.	7. Perform DMA transfer.
AST_COM	8. Execute completion AST, and check for errors.	8. Execute completion AST, and check for errors.

Example 3–1 DR11–W/DRV11-WA Program Example (XAMESSAGE.MAR)

```
        .TITLE          XAMESSAGE
        .IDENT          'V04-001'
;******************************************************************************
;*                                                                            *
;*  COPYRIGHT (c) 1978, 1980, 1982, 1984 BY                                   *
;*  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                    *
;*  ALL RIGHTS RESERVED.                                                      *
;*                                                                            *
;*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED     *
;*  ONLY IN  ACCORDANCE WITH  THE  TERMS  OF  SUCH  LICENSE  AND WITH THE     *
;*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR  ANY  OTHER     *
;*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY     *
;*  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE IS  HEREBY     *
;*  TRANSFERRED.                                                              *
;*                                                                            *
;*  THE INFORMATION IN THIS SOFTWARE IS  SUBJECT TO CHANGE WITHOUT NOTICE     *
;*  AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT     *
;*  CORPORATION.                                                              *
;*                                                                            *
;*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY OF ITS     *
;*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.                   *
;*                                                                            *
;*                                                                            *
;******************************************************************************
;
;
;++
;
; ABSTRACT:
;
;       This module allows you to connect a DR11-W to a DRV11-WA; or
;       a DR11-W to another DR11-W in an interprocessor link and to
;       perform data transfers from one processor to the other.
;
```

(Continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11-WA Program Example (XAMESSAGE.MAR)

```
;--
        .SBTTL          LOCAL DEFFINITIONS AND STORAGE
;++
; XAMESSAGE ROUTINE
;
; CALLING SEQUENCE:
;
;       CALL (BUFFER_ADDRESS,BUFFER_SIZE,TRANSFER_DIRECTION,CHANNEL,-
;             EVENT_FLAG,TIME_OUT,STATUS_ADDRESS,LOCAL_DEVICE,REMOTE_DEVICE)
;
;               BUFFER_ADDRESS = ADDRESS OF DATA BUFFER TO TRANSFER
;               BUFFER_SIZE = SIZE IN BYTES OF DATA BUFFER TO TRANSFER.
;                 NOTE THAT RECEIVER AND TRANSMITTER MUST AGREE ON THE
;                 SIZE OF THE TRANSFER.
;               TRANSFER_DIRECTION = DIRECTION FOR DATA TO GO
;                       0 = TRANSMIT
;                       1 = RECEIVE
;               CHANNEL = CHANNEL ASSIGNED TO DEVICE (DR11-W OR DRV11-WA)
;               EVENT_FLAG = EVENT FLAG TO SET WHEN TRANSFER COMPLETE
;               TIME_OUT = I/O TIME-OUT VALUE IN SECONDS
;               STATUS_ADDRESS = ADDRESS OF 20 BYTE ARRAY TO RECEIVE
;                 FINAL STATUS - ONLY FILLED IN IF USER'S PARAMETERS ARE
;                 ALL VALID.
;                       IOSB - 8 BYTES
;                               I/O STATUS BLOCK FROM QUEUE I/O REQUEST
;                       ERROR - 4 BYTES - NOT USED - FOR COMPATIBILITY
;                               WITH OLD VERSIONS OF THIS MODULE.
;                       STATE - 4 BYTES
;                               THIS FIELD TRACKS WHICH QIO WAS THE LATEST
;                               ONE TO BE PERFORMED.
;                               01 - LAST QIO WAS ONE IN THE MAIN ROUTINE.
;                               02 - LAST QIO WAS ONE IN AST_GO.
;                       SSRV_STS - 4 BYTES
;                               VALUE OF R0 RETURNED FROM THE LAST SYSTEM
;                               SERVICE EXECUTED.
;             LOCAL_DEVICE = TYPE OF DEVICE AT LOCAL END OF LINK.
;                       DR11_W = 1
;                       DRV11_WA = 2
;             REMOTE_DEVICE = TYPE OF DEVICE AT REMOTE END OF LINK.
;                       DR11_W = 1
;                       DRV11_WA = 2
;
;--
```

(Continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11-WA Program Example (XAMESSAGE.MAR)

```
        $SSDEF
; PARAMETER OFFSETS.
BUFFER_P = 4
BUF_SIZE_P = 8
DIRECTION_P = 12
CHAN_P = 16
EFN_P = 20
TIME_P = 24
STS_ADDR_P = 28
LCL_DEVICE_P = 32
REM_DEVICE_P = 36
        .PSECT        XADATA,LONG
; SAVED PARAMETER VALUES.
BUFFER:         .LONG       0           ; SAVED BUFFER ADDRESS
BUF_SIZE:       .LONG       0           ; SAVED BUFFER SIZE
DIRECTION:      .LONG       0           ; DIRECTION OF TRANSFER
CHAN:           .LONG       0           ; SAVED CHANNEL ASSIGNED TO DR11-W
EFN:            .LONG       0           ; SAVED EVENT FLAG NUMBER
TIME:           .LONG       0           ; SAVED TIME-OUT VALUE
STS_ADDR:       .LONG       0           ; ADDRESS OF CALLERS STATUS VARIABLE
; DEFINE DEVICE TYPES AT BOTH ENDS OF INTERPROCESSOR LINK.
DR11_W = 1
DRV11_WA = 2
LCL_DEVICE:     .BLKL       1           ; TYPE OF DEVICE ON THIS SYSTEM.
REM_DEVICE:     .BLKL       1           ; TYPE OF DEVICE AT OTHER
                                        ;   END OF LINK.
AST:            .BLKL       1
; NOTE - ORDER IS ASSUMED FOR NEXT FOUR VARIABLES
IOSB:           .QUAD       0           ; QIO IOSB
ERROR:          .LONG       0           ; ERROR VALUE PARAMETER
STATE:          .LONG       0           ; STATE VARIABLE
SSRV_STS:       .LONG       0           ; SYSTEM SERVICE STATUS
        .PAGE
        .SBTTL    VALIDATE AND SAVE CALLER'S PARAMETERS
        .PSECT    XACODE,NOWRT
```

(Continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11-WA Program Example (XAMESSAGE.MAR)

```
        .ENTRY    XAMESSAGE,^M<R2,R3,R4,R5>
; VALIDATE AND SAVE CALLER'S PARAMETERS
        CLRQ      W^IOSB                    ; CLEAR IOSB
        CLRL      W^ERROR                   ; CLEAR ERROR FIELD
        CLRL      W^SSRV_STS                ; CLEAR SYS SERVICE RETURN STATUS.
        CMPW      (AP),#9                   ; MUST HAVE 9 PARAMETERS
        BEQL      10$                       ; BR IF OKAY
        BRW       BADPARAM                  ; BR TO SIGNAL ERROR
10$:    MOVL      BUFFER_P(AP),W^BUFFER     ; GET BUFFER ADDRESS
        MOVL      @BUF_SIZE_P(AP),W^BUF_SIZE ; GET BUFFER SIZE
        BNEQ      20$                       ; BR IF OKAY
        BRW       BADPARAM                  ; TRANSFER SIZE IS NON ZERO -- ILLEGAL
20$:    MOVZBL    @DIRECTION_P(AP),W^DIRECTION ; GET TRANSFER DIRECTION FLAG
        CMPL      W^DIRECTION,#2            ; THE ONLY LEGAL VALUES ARE 0,1
        BLEQU     25$                       ; BR IF OKAY
        BRW       BADPARAM                  ; ELSE BR TO SIGNAL ERROR
25$:    MOVL      @CHAN_P(AP),W^CHAN        ; FETCH CHANNEL
        MOVL      @EFN_P(AP),W^EFN          ; AND EVENT FLAG
        BEQL      BADPARAM                  ; MUST SPECIFY EVENT FLAG
        MOVL      @TIME_P(AP),W^TIME        ; FETCH TIME-OUT VALUE
        BNEQ      30$                       ; IF NONZERO, USE IT.
        MOVZBL    #5,W^TIME                 ; ELSE USE SOME "REASONABLE" VALUE
30$:    MOVL      STS_ADDR_P(AP),W^STS_ADDR ; GET ADDRESS OF STATUS ARRAY
        BEQL      BADPARAM                  ; IF NOT SPECIFIED, ERROR
        CLRL      @W^STS_ADDR               ; INITIALIZE STATUS VALUE
        MOVZBL    @LCL_DEVICE_P(AP),W^LCL_DEVICE ; GET LOCAL DEVICE TYPE
        CMPL      #DRV11_WA,W^LCL_DEVICE    ; IS LOCAL DEVICE A DRV11-WA?
        BEQLU     35$                       ; BRANCH IF SO.
        CMPL      #DR11_W,W^LCL_DEVICE      ; IS LOCAL DEVICE A DR11-W?
        BNEQU     BADPARAM                  ; ERROR IF IT'S NOT EITHER.
35$:    MOVZBL    @REM_DEVICE_P(AP),W^REM_DEVICE ; GET REMOTE DEVICE TYPE
        CMPL      #DRV11_WA,W^REM_DEVICE    ; IS REMOTE DEVICE A DRV11-WA?
        BEQLU     50$                       ; BRANCH IF SO.
        CMPL      #DR11_W,W^REM_DEVICE      ; IS REMOTE DEVICE A DR11-W?
        BNEQU     BADPARAM                  ; ERROR IF IT'S NOT EITHER.
50$:    $CLREF_S  EFN=EFN                   ; MAKE SURE EFN IS CLEAR
        BLBS      R0,100$                   ; BR IF NO SYS SERVICE ERROR
        RET
100$:   CMPL      #DRV11_WA,W^LCL_DEVICE    ; DISPATCH BASED ON LOCAL
                                            ;   DEVICE TYPE
        BEQL      DRV11_WA_START            ; LOCAL DEVICE IS DRV11-WA
        BRW       DR11_W_START              ; LOCAL DEVICE IS DR11-W
BADPARAM:
        MOVZWL    #SS$_BADPARAM,R0          ; ELSE RETURN ERROR.
        RET
```

(Continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11-WA Program Example (XAMESSAGE.MAR)

```
        .PAGE
        .SBTTL    START MESSAGE PROCESSOR
DRV11_WA_START:                             ; THE LOCAL DEVICE IS A DRV11-WA
        BLBC      W^DIRECTION,10$           ; BRANCH IF IT'S A TRANSMIT
                                            ;   OPERATION
        MOVAL     W^AST_COMPLETION,W^AST    ; AST_COMPLETION IS THE AST FOR
                                            ;   RECEIVE
        BRB       20$
10$:    MOVAL     W^AST_GO,W^AST            ; AST_GO IS THE AST FOR TRANSMIT
                                            ;   OPERATION
20$:    MOVL      #01,W^STATE               ; STATE = 1 => LAST QIO WAS IN MAIN
                                            ;   ROUTINE.
        $QIO_S    CHAN=W^CHAN,-             ; BLOCK MODE READ - EVEN IF IT'S
                  FUNC=#<IO$_READLBLK!IO$M_TIMED!IO$M_SETFNCT>,- ; TRANSMIT
                  IOSB=W^IOSB,-
                  ASTADR=@W^AST,-
                  P1=@W^BUFFER,-            ; ADDRESS OF CALLER'S DATA BUFFER
                  P2=W^BUF_SIZE,-           ; LENGTH OF DATA BUFFER
                  P3=W^TIME,-               ; TIMEOUT VALUE
                  P4=#7                     ; INTERRUPT+READ
        BRW       MAIN_EXIT                 ; EXIT MAIN ROUTINE.
DR11_W_START:                               ; LOCAL DEVICE IS DR11-W
        MOVL      #01,W^STATE               ; STATE = 1 => LAST QIO WAS IN MAIN
                                            ;   ROUTINE.
        $QIO_S    CHAN=W^CHAN,-             ; QIO TO ENABLE AST'S
                  FUNC=#<IO$_SETMODE!IO$M_ATTNAST>,-
                  IOSB=W^IOSB,-
                  P1=W^AST_GO
        BLBC      R0,MAIN_EXIT              ; BRANCH ON ERROR - ALL DONE.
        BLBS      W^DIRECTION,MAIN_EXIT     ; BRANCH IF THIS IS A RECEIVE
                                            ;   OPERATION
        CMPL      #DR11_W,W^REM_DEVICE      ; IS REMOTE DEVICE A DR11-W?
        BNEQU     MAIN_EXIT                 ; BRANCH IF NOT.
        $QIO_S    CHAN=W^CHAN,-             ; PERFORM 0-LENGTH QIO.  THIS
                  FUNC=#<IO$_WRITELBLK!IO$M_SETFNCT>,- ; SERVES TO SET THE
                  IOSB=W^IOSB,-             ; FNCT BITS (CONTAINED IN P4),
                  P1=@W^BUFFER,-            ; IN THE CSR, INTERRUPTING THE
                                            ;   REMOTE DR11-W.
                  P2=#0,-
                  P4=#2
MAIN_EXIT:
        MOVL      R0,W^SSRV_STS             ; SAVE QIO STATUS RETURN
        MOVC3     #20,W^IOSB,@W^STS_ADDR    ; RETURN STATUS TO THE USER
        BLBS      W^SSRV_STS,10$            ; IF SUCCESS, DON'T SET EVFLAG YET
        $SETEF_S  EFN=W^EFN                 ; IF ERROR, SET EVENT FLAG
                                            ;    -- ALL DONE.
10$:    MOVL      W^SSRV_STS,R0             ; RESTORE R0 STATUS RETURN.
        RET
```

(Continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11-WA Program Example (XAMESSAGE.MAR)

```
        .PAGE
        .SBTTL   AST_GO - AST WHICH INITIATES THE QIO TO PERFORM ACTUAL TRANSFER.
        .ENTRY   AST_GO,^M<R2,R3,R4,R5>
;
; This AST is called to perform the $QIO which begins the actual transfer
; of user data.  (Hence the name AST_GO.)
;
        BLBS     W^DIRECTION,AST_RECEIVE     ; BRANCH IF RECEIVE OPERATION
;
; On a DR11-W, this AST is delivered as a result of an interrupt from the
; remote device, so no status checking is necessary.  On a DRV11-WA, this AST
; is delivered as a result of an intentionally premature I/O completion, so
; we expect the status return to be SS$_OPINCOMPL.
;
AST_XMIT:
        CMPL     #DRV11_WA,W^LCL_DEVICE      ; IS LOCAL DEVICE A DRV11-WA?
        BNEQ     20$                         ; BRANCH IF NOT.
        CMPW     W^IOSB,#SS$_OPINCOMPL       ; STATUS SHOULD BE SS$_OPINCOMPL.
        BEQL     20$                         ; BR IF EXPECTED STATUS
        BRW      IO_DONE                     ; ELSE ERROR
20$:    MOVL     #02,W^STATE                 ; STATE = 2 => LAST QIO WAS IN
                                             ;   AST_GO.
        $QIO_S   CHAN=W^CHAN,-               ; BLOCK MODE WRITE
                 FUNC=#<IO$_WRITELBLK!IO$M_TIMED!IO$M_SETFNCT!IO$M_CYCLE>,-
                 IOSB=W^IOSB,-
                 ASTADR=W^AST_COMPLETION,-
                 P1=@W^BUFFER,-              ; ADDRESS OF CALLER'S DATA BUFFER
                 P2=W^BUF_SIZE,-             ; LENGTH OF BUFFER
                 P3=W^TIME,-                 ; TIMEOUT VALUE
                 P4=#4                       ; FNCT BITS FOR CSR
        BLBS     R0,40$                      ; RETURN IF QIO STARTED OK
        BRW      IO_DONE                     ; ALL DONE IF ERROR OCCURRED.
40$:    RET                                  ; DISMISS THIS AST, AND
                                             ;   WAIT FOR AST_COMPLETION
;
; AST_RECEIVE is only used by the DR11-W, since the DRV11-WA initiates
; the actual data transfer from the main routine when it is the receiver.
;
AST_RECEIVE:
        MOVL     #02,W^STATE                 ; STATE = 2 => LAST QIO WAS IN
                                             ;   AST_GO.
        $QIO_S   CHAN=W^CHAN,-               ; BLOCK MODE READ
                 FUNC=#<IO$_READLBLK!IO$M_TIMED!IO$M_SETFNCT>,-
                 IOSB=W^IOSB,-
                 ASTADR=W^AST_COMPLETION,-   ; ADDRESS OF AST FOR I/O COMPLETION
                 P1=@W^BUFFER,-              ; ADDRESS OF CALLER'S DATA BUFFER
                 P2=W^BUF_SIZE,-             ; LENGTH OF DATA BUFFER
                 P3=W^TIME,-                 ; TIMEOUT VALUE
                 P4=#7                       ; INTERRUPT+READ
        BLBS     R0,10$                      ; RETURN IF QIO STARTED OK
        BRW      IO_DONE                     ; ON ERROR, WE'RE ALL DONE.
10$:    RET
```

(Continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11-WA Program Example (XAMESSAGE.MAR)

```
        .PAGE
        .SBTTL   AST_COMPLETION - COMPLETION ROUTINE FOR I/O TRANSFER.
        .ENTRY   AST_COMPLETION,^M<R2,R3,R4,R5>

;
; This AST is called when the actual transfer of data is complete.  Note that
; the status value in the IOSB must be checked by the caller when we're done.
; IO_DONE is also called when an error occurs and the handshaking sequence
; must be terminated.
;
IO_DONE:
        MOVC3    #20,W^IOSB,@W^STS_ADDR      ; RETURN STATUS TO THE USER
        $SETEF_S EFN=W^EFN                   ; SET THE CALLER'S EVENT FLAG
        MOVZBL   #SS$_NORMAL,R0              ; SIGNAL SUCCESSFUL AST COMPLETION.
        RET
        .END
```

4 DR32 Interface Driver

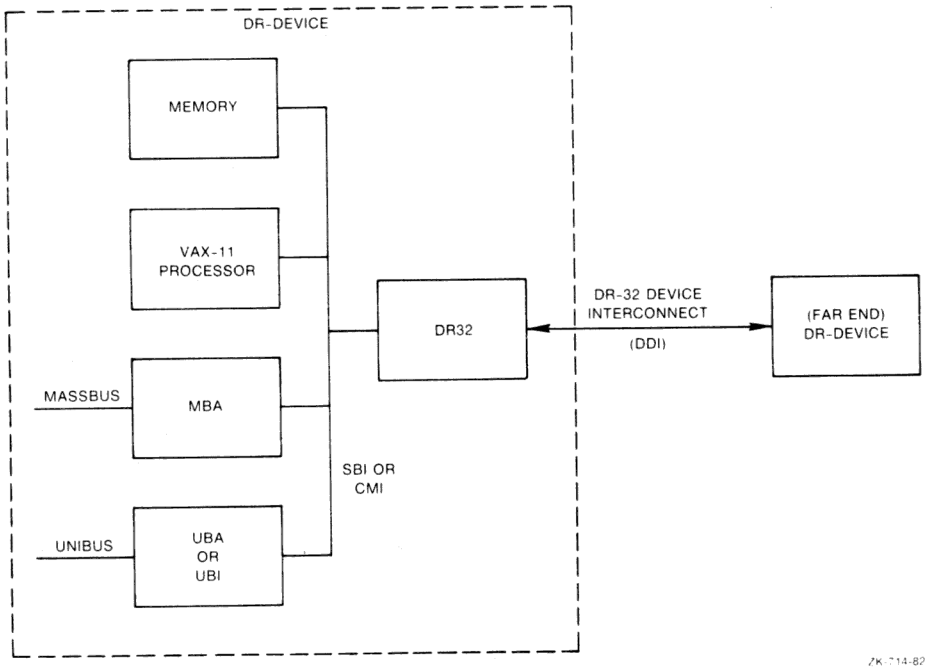
This section describes the use of the VAX/VMS DR32 interface driver.

4.1 Supported Device

The DR32 is an interface adapter that connects the internal memory bus of a VAX processor to a user-accessible bus called the DR32 device interconnect (DDI). Two DR32s can be connected to form a VAX processor-to-processor link (non-DECnet). Figure 4–1 shows the relationship of the DR32 to a VAX/VMS system and the DDI.

As a general-purpose data port, the DR32 is capable of moving continuous streams of data to or from memory at high speed. Data from a user device to disk storage must go through an intermediate buffer in physical memory.

Figure 4–1 Basic DR32 Configuration



4.1.1 DR32 Device Interconnect

The DR32 device interconnect (DDI) is a bidirectional path for the transfer of data and control signals. Control signals sent over the DDI are asynchronous and interlocked; data transfers are synchronized with clock signals. Any connection to the DDI is called a *DR-device*. The DDI provides a point-to-point connection between two DR-devices, one of which must be a VAX

processor. The DR-device connected to the external end of the DDI is called the *far-end DR-device.*

4.2 DR32 Features and Capabilities

The DR32 driver provides the following features and capabilities:

- 32-bit parallel data transfers
- High bandwidth (6 megabytes/second on the DDI with a VAX–11/780 or 3.12 megabytes/second on a VAX–11/750)
- Word or byte alignment of data
- Half-duplex operation
- Hardware-supported (I/O driver-independent) memory mapping
- Separate control and data interconnects
- Command and data chaining
- Direct software link between the DR32 and the user process
- Synchronization of the user program with DR32 data transfers
- Transfers initiated by an external device

The following sections describe command and data chaining, data transfers, power failure, and interrupts.

4.2.1 Command and Data Chaining

Command chaining is the execution of commands without software intervention for each command. Commands are chained in the sense that they follow each other on a queue. After a QIO function starts the DR32, any number of DR32 commands can be executed during that QIO operation. This process continues until either the transfer is halted (a command packet is fetched that specifies a halt command) or an error occurs. (Section 4.4.3 describes command packets.)

Command packets can specify data chaining. In data chaining, a number of physical memory buffers appear as one large buffer to the far-end DR-device. Data chaining is completely transparent to this device; transfers are seen as a continuous stream of data. Chained buffers can be of arbitrary byte alignment and length. The length of a transfer appears to the far-end DR-device as the total of all the byte counts in the chain, and since chains in the DR32 can be of unlimited length, the device interprets the byte count as potentially infinite.

4.2.2 Far-End DR-Device Initiated Transfers

For the far-end DR-device, the DR32 provides the capability of initiating data transfers to memory, that is, of initiating random access mode. This mode is used when two DR-32s are connected to form a processor-to-processor link. Random access consists of data transfers to or from memory without notification of the VAX processor. Random access can be discontinued either by specifying a command packet with random access disabled or by an abort operation from either the controlling process or the far-end DR-device.

4.2.3 Power Failure

If power fails on the DR32 interface but not on the system, the DR32 driver aborts the active data transfer and returns the status code SS$_POWERFAIL in the I/O status block. If a system power failure occurs, the DR32 driver completes the active data transfer when power is recovered and returns the status code SS$_POWERFAIL.

4.2.4 Interrupts

The DR32 interface can interrupt the DR32 driver for any of the following reasons:

- An abort has occurred. The QIO operation is completed.
- A DR32 power-down or power-up sequence has occurred.
- An unsolicited control message has been sent to the DR32. If this command packet's interrupt control field is properly set up, a packet AST interrupt occurs. The interrupt occurs after the command packet obtained from the free queue (FREEQ) is placed on the termination queue (TERMQ).
- The DR32 enters the halt state. The QIO operation is completed.
- A command packet that specifies an unconditional interrupt has been placed onto TERMQ. The result is a packet AST.
- A command packet with the "interrupt when TERMQ empty" bit set was placed on an empty TERMQ. The result is a packet AST.

4.3 Device Information

Users can obtain information on DR32 characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume*.)

$GETDVI returns DR32 characteristics when you specify the item code DVI$_DEVCHAR. Table 4–1 lists these characteristics, which are defined by the $DEVDEF macro.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and class names, which are defined by the $DCDEF macro. The device type is DT$_DR780 for the DR780 and DT$_DR750 for the DR750. The device class for the DR32 is DC$_REALTIME. DVI$_DEVDEPEND returns a longword field in which the low-order byte contains the last data rate value loaded into the DR32 data rate register.

Table 4–1 DR32 Device Characteristics

Characteristic[1]	Meaning
Dynamic Bit (Conditionally Set)	
DEV$M_AVL	Device is available.
Static Bits (Always Set)	
DEV$M_IDV	Input device
DEV$M_ODV	Output device
DEV$M_RTM	Real-time device

[1]Defined by the $DEVDEF macro.

4.4 Programming Interface

The DR32 interface is supported by a device driver, a high-level language procedure library of support routines, and a program for microcode loading.

After issuing an IO$_STARTDATA request to the DR32 driver, application programs communicate directly with the DR32 interface by inserting command packets onto queues. This direct link between the application program and the DR32 interface provides faster communication by avoiding the necessity of going through the I/O driver.

Two interfaces are provided for accessing the DR32: a QIO interface and a support routine interface. The QIO interface requires that the application program build command packets and insert them onto the DR32 queues. The support routine interface, on the other hand, provides procedures for these functions and, in addition, performs housekeeping functions, such as maintaining command memory.

The support routine interface was designed to be called from high-level languages, such as FORTRAN, where the data manipulation required by the QIO interface might be awkward. Note, however, that the user of the support routines interface must be as knowledgeable about the DR32 and the meaning of the fields in the command packets as the user of the QIO interface.

4.4.1 DR32—Application Program Interface

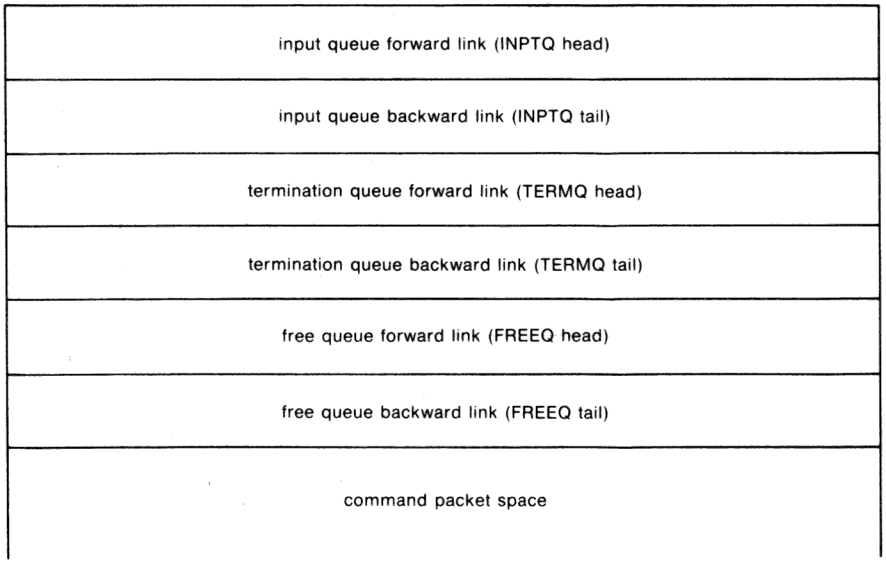
Application programs interface with the DR32 through two memory areas. These areas are called the *command block* and the *buffer block*. The addresses and sizes of the blocks are determined by the application program and are passed to the DR32 driver as arguments to the IO$_STARTDATA function, which starts the DR32 (see Section 4.4.5.2).

Both blocks are locked into memory while the DR32 is active. The buffer block defines the area of memory that is accessible to the DR32 for the transfer of data between the far-end DR-device and the DR32. The command block contains the headers for the three queues that provide the communication path between the DR32 and the application program, and space in which to build command packets.

The interface between the DR32 and the application program contains three queues: the input queue (INPTQ), the termination queue (TERMQ), and the free queue (FREEQ). Information is transferred between the DR32 and the far-end DR-device through command packets. The three queue structures control the flow of command packets to and from the DR32. The application program builds a command packet and inserts it onto INPTQ. The DR32 removes the packet, executes the specified command, enters some status information, and then inserts the packet onto TERMQ. Unsolicited input from the far-end DR-device is placed in packets removed from FREEQ and inserted onto TERMQ.

The INPTQ, TERMQ, and FREEQ headers are located in the first six longwords of the command block. Since the queues are self-relative, that is, they use the VAX/VMS self-relative queue instructions (INSQHI, INSQTI, REMQHI, and REMQTI), the headers must be quadword-aligned. The application program must initialize all queue headers. Figure 4–2 shows the position of the queue headers in the command block. Section 4.4.2 describes queue processing in greater detail.

Figure 4–2 Command Block (Queue Headers)



4.4.2 Queue Processing

Three queue structures control the flow of command packets to and from the DR32:

- Input queue (INPTQ)
- Termination queue (TERMQ)
- Free queue (FREEQ)

The DR32 removes command packets from the heads of FREEQ and INPTQ and inserts command packets onto the tail of TERMQ. For command sequences initiated by the application program, the DR32 removes command packets from the head of INPTQ, processes them, and returns them to the tail of TERMQ. Queue processing is performed by the DR32 with the equivalent of the INSQTI and REMQHI instructions. To remove a packet from INPTQ, the DR32 executes the equivalent of REMQHI HDR, CMDPTR where CMDPTR is a DR32 register used as a pointer to the current command packet and HDR specifies the INPTQ header. To insert a packet onto TERMQ, the DR32 executes the equivalent of INSQTI CMDPTR, HDR. The user process performs similar operations with the queues, inserting packets onto the head or tail of INPTQ and normally removing packets from the head of TERMQ.

If any of the queues are currently being accessed by the DR32, the program's interlocked queue instructions will fail for either of the following reasons:

- The DR32 is currently removing a packet from INPTQ or FREEQ, or inserting a packet onto TERMQ, and the operation will be completed shortly.
- The DR32 detects an error condition, for example, an unaligned queue, that prevents it from completing the queue operation. In this case, the transfer is aborted and the I/O status block contains the error that caused the abort.

To distinguish between these two conditions, the application program must include a queue retry mechanism that retries the queue operation a reasonable number of times (for example, 25) before determining that an error condition exists. An example of a queue retry mechanism is shown in the program example (Program B in Section 4.7).

If the DR32 discerns that any of the queues are interlocked, it retries the operation until it completes or the DR32 is aborted.

4.4.2.1 Initiating Command Sequences

If a command packet is inserted onto an empty INPTQ, the application program must notify the DR32 of this event. This is done by setting bit 0, the GO bit, in a DR32 register. The IO$_STARTDATA function returns the GO bit's address to the application program. After notification by the GO bit that there are command packets on its INPTQ, the DR32 continues to process the packets until INPTQ is empty.

The GO bit can be safely set at any time. While processing command packets, the DR32 ignores the GO bit. If the GO bit is set when the DR32 is idle, the DR32 will attempt to remove a command packet from INPTQ. If INPTQ is empty at this time, the DR32 clears the GO bit and returns to the idle state.

4.4.2.2 Device-Initiated Command Sequences

If the DR-device that interfaces the far-end of the DDI is capable of transmitting unsolicited control messages, messages of this type can be transmitted to the local DR32. These messages are not synchronized to the application program command flow. Therefore, the DR32 uses a third queue, FREEQ, to handle unsolicited messages. Normally, the application program inserts a number of empty command packets onto FREEQ to allow the external device to transmit control messages.

If a control message is received from the far-end DR-device, the DR32 removes an empty command packet from the head of FREEQ, fills the device message field of this packet with the control message and, when the transmission is completed, inserts the packet onto the tail of TERMQ. (The

device message field in this command packet must be large enough for the entire message or a length error will occur.) The application program then removes the packet from TERMQ. If the command packet is from FREEQ, the XF$M_PKT_FREQPK bit in the DR32 status longword is set.

4.4.3 Command Packets

To provide for direct communication between the controlling process and the DR32, the DR32 fetches commands from user-constructed command packets located in physical memory. Command packets contain commands for the DR32, such as the direction of transfer, and messages to be sent to the far-end DR-device. The DR32 is simply the conveyer of these messages; it does not examine or add to their content. The controlling process builds command packets and manipulates the three queues, using the four VAX self-relative queue instructions. Figure 4–3 shows the DR32 queue flow. Figure 4–4 shows the contents of a DR32 command packet.

Figure 4–3 DR32 Command Packet Queue Flow

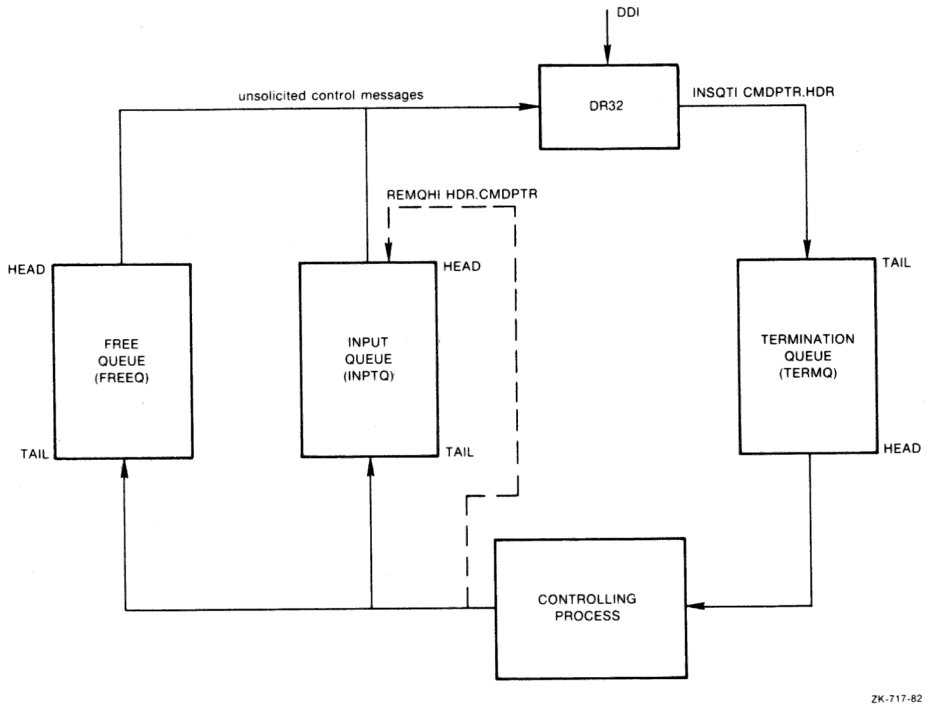
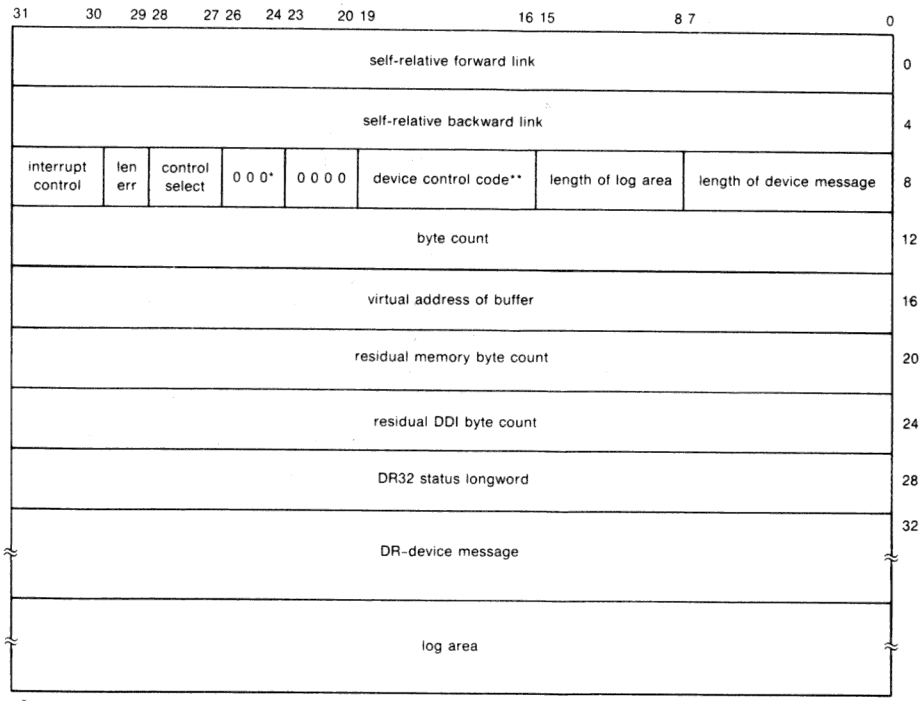


Figure 4–4 DR32 Command Packet

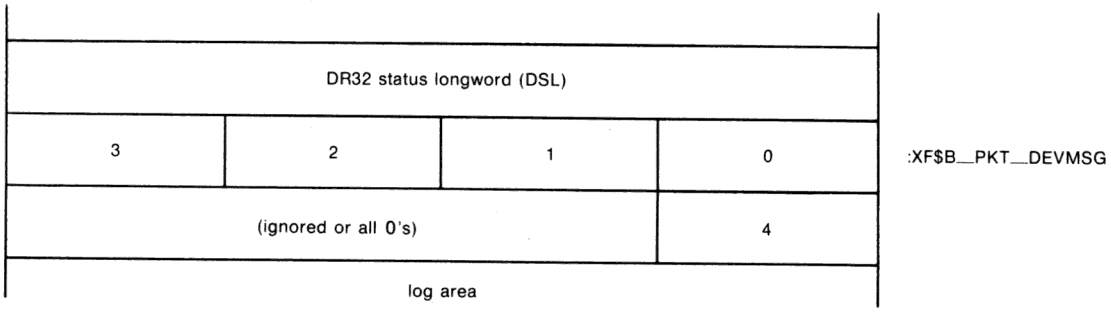


* Bits 31:24 = Packet Control Byte
**Bits 23:16 = Command Control Byte

ZK-718-82

4.4.3.1 Length of Device Message Field

The length of device message field describes the length of the DR-device message in bytes. The message length must be less than 256 bytes. Note, however, that the length of device message field itself must always be an integral number of quadwords long. For example, if the application program requires a five-byte device message, it must write a 5 in the length of device message field, but allocate eight bytes for the device message field itself. In this case, the last three bytes of the field are ignored by the DR32 when transmitting a message, or written as zeros when receiving a message:



ZK-719-82

The symbolic offset for the length of device message field is XF$B_PKT_MSGLEN.

4.4.3.2 Length of Log Area Field

The length of log area field describes the length of the log area in bytes. The length specified must be less than 256 bytes. Note, however, that the length of log area field itself must be an integral number of quadwords long. For example, if the application program requires a five-byte log area field, it must write a 5 in the length of log area field, but allocate eight bytes for the log area field itself. In this case, the last three bytes of the field are written as zeros when receiving a log message (log messages are always received). The symbolic offset for the length of log area field is XF$B_PKT_LOGLEN.

4.4.3.3 Device Control Code Field

The device control field describes the function performed by the DR32. The field occupies the lower half of the command control byte (bits 16 through 23). The VAX/VMS operating system defines the following values:

Symbol	Value	Function
XF$K_PKT_RD	0	Read device
XF$K_PKT_RDCHN	1	Read device chained
XF$K_PKT_WRT	2	Write device
XF$K_PKT_WRTCHN	3	Write device chained
XF$K_PKT_WRTCM	4	Write device control message
	5	(reserved)
XF$K_PKT_SETTST	6	Set self-test
XF$K_PKT_CLRTST	7	Clear self-test
XF$K_PKT_NOP	8	No operation
XF$K_PKT_DIAGRI	9	Diagnostic read internal
XF$K_PKT_DIAGWI	10	Diagnostic write internal
XF$K_PKT_DIAGRD	11	Diagnostic read DDI
XF$K_PKT_DIAGWC	12	Diagnostic write control message
XF$K_PKT_SETRND	13	Set random enable
XF$K_PKT_CLRRND	14	Clear random enable
XF$K_PKT_HALT	15	Set halt

Table 4–2 describes the functions performed by the different device control codes.

Table 4–2 Device Control Code Descriptions

Function	Meaning
Read device	This function specifies a data transfer from the far-end DR-device to the DR32. The control select field (see Section 4.4.3.4) describes the information to be transferred prior to the initiation of the data transfer.
Read device chained	This function specifies a data transfer from the far-end DR-device to the DR32. The DR32 chains data to the buffer specified in the next command packet in INPTQ. A command packet that specifies the read device chained function must be followed by a command packet that specifies either the read device chained function or the read device function. All other device control codes cause an abort. If a read device chained function is specified, the chain continues. However, if a read device function is specified, that command packet is the last packet in the chain.
Write device and write device chained	These functions specify data transfers from the DR32 to the far-end DR-device. Otherwise, they are similar to read device and read device chained functions.
Write device control message	This function specifies the transfer of a control message to the far-end DR-device. This message is contained in the device message field of this command packet. The write device control message function directs the controlling DR32 to ignore the byte count and virtual address fields in this command packet.
Set self-test	This function directs the DR32 to set an internal self-test flag and to set a disable signal on the DDI. This signal informs the far-end DR-device that the DR32 is in self-test mode. While in self-test mode, the DR32 can no longer communicate with the far-end DR-device.
Clear self-test	This function directs the DR32 to clear the internal self-test flag set by the set self-test function and to return to the normal mode of operation.
No operation	This function explicitly does nothing.
Diagnostic read internal	This function directs the DR32 to fill the memory buffer, which is described by the virtual address and byte count specified in the current command packet, with the data that is stored in the DR32 data silo. The buffer is filled in a cyclical manner. For example, on the DR780 every 128-byte section of the buffer receives the silo data. The amount of data stored in the buffer equals the DDI byte count minus the SBI byte count. The DDI byte count is equal to the original byte count. No data transmission takes place on the DDI for this function. On the DR780, the diagnostic read internal function destroys the first four bytes in the silo before storing the data in the buffer.

Table 4–2 (Cont.) Device Control Code Descriptions

Function	Meaning
Diagnostic write internal	This function, together with the diagnostic read internal function, is used to test the DR32 read and write capability. The diagnostic write internal function directs the DR32 to store data, which is contained in the memory buffer described by the current command packet, in the DR32 data silo, a FIFO-type buffer. No data transmission takes place on the DDI for this function. The diagnostic write internal function terminates when either of the following conditions occurs: • The memory buffer is empty (the SBI byte count is 0). • An abort has occurred. When the function terminates, the amount of data in the silo equals the DDI byte count minus the SBI memory byte count. (Sections 4.4.3.9 and 4.4.3.10 describe these values.)
Diagnostic read DDI	This function tests transmissions over the data portion of the DDI. The DR32 must be in the self-test mode. If not, an abort will occur. On the DR780, the diagnostic read DDI function transmits the contents of DR32 data silo locations 0 to 127 over the DDI and returns the data to the same locations. If data transmission is normal, that is, without errors, the residual memory count is equal to the original byte count, the residual DDI count is 0, and the contents of the silo remain unchanged.
Diagnostic write control message	This function tests transmissions over the control portion of the DDI. The DR32 must be in self-test mode. If not, an abort will occur. The diagnostic write control message function directs the DR32 to remove the command packet on FREEQ and check the length of message field. Then the first byte of the message in the command packet on INPTQ is transmitted and read back on the control portion of the DDI. This byte is then written into the message space of the packet from FREEQ. The updated packet from FREEQ is inserted onto TERMQ and is followed by the packet from INPTQ.

Table 4–2 (Cont.) Device Control Code Descriptions

Function	Meaning
Set random enable and clear random enable	This function directs the DR32 to accept read and write commands sent by the far-end DR-device. Range-checking is performed to verify that all addresses specified by the far-end DR-device for access are within the buffer block. Far-end DR-device initiated transfers to or from the VAX memory are conducted without notification of the VAX processor or the application program. The clear random enable function directs the DR32 to reject far-end DR-device-initiated transfers. Random access mode must be enabled when the DR32 is used in a processor-to-processor link.
Set halt	This function places the DR32 in a halt state. The set halt function always generates a packet interrupt regardless of the value in the interrupt control field (see Section 4.4.3.6). If an AST routine was requested on completion of the QIO function (see Sections 4.4.5.2 and 4.4.6.2), the routine is called after the command packet containing the set halt function has been processed by the DR32.

The following symbolic offsets are defined for the device control code field:

Symbol	Meaning
XF$B_PKT_CMDCTL	Byte offset from the beginning of the command packet
XF$V_PKT_FUNC	Bit offset from XF$B_PKT_CMDCTL
XF$S_PKT_FUNC	Size of the device control code bit field

4.4.3.4 Control Select Field

This field describes the part of the command packet that will be transmitted to the far-end DR-device. The control select field is examined only for the read device, read device chained, write device, and write device chained functions; for all others, it is ignored. The VAX/VMS operating system defines the following values:

Symbol	Value	Function
XF$K_PKT_NOTRAN	0	No transmission. Nothing is transmitted over the control portion of the DDI. However, if the command packet specifies a data transfer, data can be transmitted over the data portion of the DDI. The primary use of this code is during data chaining.

Symbol	Value	Function
XF$K_PKT_CB	1	Command control byte (bits 23:16) only. This code directs the DR32 to transmit the contents of the command control byte, which includes the device control code field, to the far-end DR-device. This code is used primarily at the start of data chain or nondata chain commands.
XF$K_PKT_CBDM	2	Command control byte and device message. This code directs the DR32 to transmit the command control byte, and then the device message. It is used primarily when an interface requires more than one byte of command.
XF$K_PKT_CBDMBC	3	Command control byte, device message, and byte count. This code directs the DR32 to transmit the command control byte, the byte count, and the device message (in that order). It is used primarily during processor-to-processor link operations. In this case the device message must be exactly four bytes in length and contain the virtual address of the buffer in the far-end processor's memory.

The following symbolic offsets are defined for the control select field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_CISEL	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_CISEL	Size of control select bit field

4.4.3.5 Suppress Length Error Field

The suppress length error field function prevents the DR32 from aborting if the data transfer on the DDI is terminated by the far-end DR-device before the DDI byte counter has reached zero.

The following symbolic offsets are defined for the suppress length error field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_SLNERR	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_SLNERR	Size of the suppress length error bit field

4.4.3.6 Interrupt Control Field

The interrupt control field determines the conditions under which an interrupt is generated, on a packet-by-packet basis, when the DR32 places this command packet onto TERMQ. Depending on the conditions specified in the IO$_STARTDATA call, the interrupt can set an event flag or call an AST routine.

Symbol	Value	Function
XF$K_PKT_UNCOND	0	Interrupt unconditionally
XF$K_PKT_TMQMT	1	Interrupt only if TERMQ was previously empty
XF$K_PKT_NOINT	2,3	No interrupt

If the set halt function is active, the interrupt control field is ignored. The set halt function unconditionally causes a packet interrupt. The following symbolic offsets are defined for the interrupt control field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_INTCTL	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_INTCTL	Size of the interrupt control bit field

4.4.3.7 Byte Count Field

The byte count field specifies the size in bytes of the data buffer for this data transfer. Together with the virtual address of buffer field, this field describes the buffer in the buffer block that the DR32 will read from or write to.

The following symbolic offset is defined for the byte count field:

Symbol	Meaning
XF$B_PKT_BFRSIZ	Byte offset from the beginning of the command packet

4.4.3.8 Virtual Address of Buffer Field

The virtual address of buffer field specifies the virtual address of the data buffer for this data transfer. Together with the byte count field, this field describes the buffer in the buffer block that the DR32 will read from or write to.

The following symbolic offset is defined for the virtual address of buffer field:

Symbol	Meaning
XF$B_PKT_BFRADR	Byte offset from the beginning of the command packet

4.4.3.9 Residual Memory Byte Count Field

After completion of a read device, read device chained, write device, write device chained, diagnostic read internal, diagnostic write internal, or diagnostic read DDI function specified in this command packet, the DR32 places the packet onto TERMQ for return to the controlling process. At that time, this field will contain a byte count. The difference between the count specified in the byte count field and the count in this field is the number of bytes transferred to or from physical memory, depending on the direction of transfer.

The following symbolic offset is defined for the residual memory byte count field:

Symbol	Meaning
XF$L_PKT_RMBCNT	Byte offset from the beginning of the command packet

(See also the descriptions of the diagnostic read internal and diagnostic write internal functions in Table 4–2.)

4.4.3.10 Residual DDI Byte Count Field

After completion of a read device, read device chained, write device, write device chained, diagnostic read internal, diagnostic write internal, or diagnostic read DDI function specified in this command packet, the DR32 places the packet onto TERMQ for return to the controlling process. At this time, the residual DDI byte count field contains a byte count. The difference between the count specified in the byte count field and the count in this field is the number of bytes transferred to or from the far-end DR-device over the DDI, depending on the direction of transfer.

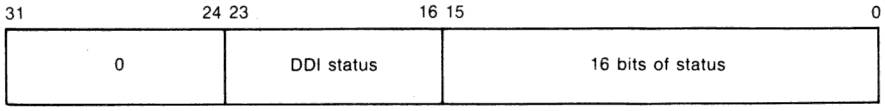
The following symbolic offset is defined for the residual DDI byte count field:

Symbol	Meaning
XF$L_PKT_RDBCNT	Byte offset from the beginning of the command packet

(See also the descriptions of the diagnostic read internal and diagnostic write internal functions in Table 4–2.)

4.4.3.11 DR32 Status Longword (DSL)

The DR32 stores the final status for a command packet in the DR32 status longword before inserting the packet onto TERMQ. The longword contains two distinct status fields:



ZK-720-82

Table 4–3 lists the names for the status bits returned in the DR32 status longword.

Table 4–3 DR32 Status Longword (DSL) Status Bits

Name	Meaning
	16 Status Bits
XF$V_PKT_SUCCESS XF$M_PKT_SUCCESS	If set, the command was performed successfully. If not set, one of the following bits must be set: XF$M_PKT_INVPTE XF$M_PKT_RNGERR XF$M_PKT_UNGERR XF$M_PKT_INVPKT XF$M_PKT_FREQMT XF$M_PKT_DDIDIS XF$M_PKT_INVDDI XF$M_PKT_LENERR XF$M_PKT_DRVABT XF$M_PKT_PARERR XF$M_PKT_DDIERR
XF$V_PKT_CMDSTD XF$M_PKT_CMDSTD	If set, the command specified in this packet was started.

Table 4–3 (Cont.) DR32 Status Longword (DSL) Status Bits

Name	Meaning
	16 Status Bits
XF$V_PKT_INVPTE XF$M_PKT_INVPTE	If set, the DR32 accessed an invalid page table entry.
XF$V_PKT_FREQPK XF$M_PKT_FREQPK	If set, this command packet was removed from FREEQ.
XF$V_PKT_DDIDIS XF$M_PKT_DDIDIS	If set, the far-end DR-device is disabled.
XF$V_PKT_SLFTST XF$M_PKT_SLFTST	If set, the DR32 is in self-test mode.
XF$V_PKT_RNGERR XF$M_PKT_RNGERR	If set, a range error occurred; that is, a user-provided address was outside the command block or buffer block.
XF$V_PKT_UNQERR XF$M_PKT_UNQERR	If set, a queue element was not aligned on a quadword boundary.
XF$V_PKT_INVPKT XF$M_PKT_INVPKT	If set, this packet was not a valid DR32 command packet.
XF$V_PKT_FREQMT XF$M_PKT_FREQMT	If set, a message was received from the far-end DR-device and FREEQ was empty.
XF$V_PKT_RNDENB XF$M_PKT_RNDENB	If set, random access mode is enabled.
XF$V_PKT_INVDDI XF$M_PKT_INVDDI	If set, a protocol error occurred on the DDI.
XF$V_PKT_LENERR XF$M_PKT_LENERR	If set, the far-end DR-device terminated the data transfer before the required number of bytes was sent; or a message was received from the far-end DR-device, and the device message field in the command packet at the head of FREEQ was not large enough to hold it.
XF$V_PKT_DRVABT XF$M_PKT_DRVABT	The I/O driver aborted the transfer. Usually the result of a Cancel I/O on Channel ($CANCEL) system service request.
XF$V_PKT_PARERR XF$M_PKT_PARERR	A parity error occurred on the data or control portion of the DDI.

Table 4–3 (Cont.) DR32 Status Longword (DSL) Status Bits

Name	Meaning
	DDI Status
XF$V_PKT_DDISTS XF$S_PKT_DDISTS	DDI status. This field is the one-byte DDI register 0 of the far-end DR-device. The following three bits are offsets to this field.
XF$V_PKT_NEXREG XF$M_PKT_NEXREG	An attempt was made to access a nonexistent register in the far-end DR-device.
XF$V_PKT_LOG XF$M_PKT_LOG	The far-end DR-device registers are stored in the log area.
XF$V_PKT_DDIERR XF$M_PKT_DDIERR	An error occurred on the far-end DR-device.

4.4.3.12 Device Message Field

The device message field contains control information to be sent to the far-end DR-device. It is used when more than one byte of command is required. The number of bytes in the device message is specified in the length of device message field (see Section 4.4.3.1). (The number of bytes allocated for the length of device message field must be rounded up to an integral number of quadwords.)

If the far-end DR-device is a DR32 that is connected to another processor, a device message can be sent only if the function specified in the device control code field of this command packet is a read device, read device chained, write device, write device chained, or write device control message.

In the case of a write device control message, the data in the device message field is treated as unsolicited input and is written into the device message field of a command packet taken from the far-end DR32's FREEQ.

In the case of a read or write (either chained or unchained) function, the only message allowed is the address of the buffer in the far-end processor that either contains or will receive the data to be transferred. This device message must be exactly four bytes in length. In this case the device message is not stored in the command packet from the far-end DR32's FREEQ, but is used by the far-end DR32 to perform the data transfer.

The device message field is also used in command packets placed on FREEQ to convey unsolicited control messages from the far-end DR-device.

The symbolic offset for the device message field is XF$B_PKT_DEVMSG.

4.4.3.13 Log Area Field

The log area field receives the return status and other information from the far-end DR-device's DDI registers. Logging must be initiated by the far-end DR-device. The presence of a log area does not automatically cause logging to occur.

If the DR32 is connected in a processor-to-processor configuration, the log area field is not used.

4.4.4 DR32 Microcode Loader

The DR32 microcode loader program XFLOADER must be executed prior to using the DR32. Running XFLOADER requires CMKRNL and LOG_IO privileges. Typically, a command to run XFLOADER is placed in the site-specific system startup file. XFLOADER locates the file containing the DR32 microcode in the following manner:

1. XFLOADER attempts to open a file using the logical name XFc$WCS, where "c" is the DR32 controller designator. For example, to load microcode on device XFA0, XFLOADER attempts to open a file with the logical name XFA$WCS.
2. If the opening procedure described in step 1 fails, XFLOADER attempts to open the file SYS$SYSTEM:XF780.ULD for a DR780, or SYS$SYSTEM:XF750.ULD for a DR750. This file specification describes the default location and filename for the DR32 microcode.

After loading microcode into all available DR32s, XFLOADER either exits or hibernates, according to the following:

- If XFLOADER was run with an ordinary RUN command, that is, RUN XFLOADER, it exits after loading microcode.
- If XFLOADER was run as a separate process, as with the following command, it hibernates after loading microcode.

```
RUN/UIC=[1,1]/PROCESS=XFLOADER SYS$SYSTEM:XFLOADER
```

  In this case, XFLOADER automatically reloads microcode into the DR32s after a power recovery.

XFLOADER performs a load microcode QIO to the DR32 driver.

4.4.5 DR32 Function Codes

The DR32 I/O functions are load microcode and start data transfer. Normally, the controlling process stops data transfers with a set halt command packet. However, the Cancel I/O on Channel ($CANCEL) system service can be used to abort data transfers and complete the I/O operation.

4.4.5.1 Load Microcode

The load microcode function resets the DR32 and loads an image of DR32 microcode. It also sets the DR32 data rate to the last specified value. Physical I/O privilege is required. The VAX/VMS operating system defines a single function code:

- IO$_LOADMCODE—load microcode

The load microcode function takes two device/function-dependent arguments:

- P1—the starting virtual address of the microcode image that is to be loaded into the DR32
- P2—the number of bytes to be loaded (maximum of 5120 for the DR780)

If any data transfer requests are active when a load microcode request is issued, the load request is rejected and SS$_DEVACTIVE is returned in the I/O status block.

The microcode is verified by addressing each microword and checking for a parity error. (The microcode is not compared to the buffer image.) If there are no parity errors, then the microcode was loaded successfully and the driver sets the microcode valid bit in one of the DR32 registers. If there is a parity error, SS$_PARITY is returned in the I/O status block. (The valid bit is cleared by the reset operation.)

In addition to SS$_PARITY, three other status codes can be returned in the I/O status block: SS$_NORMAL, SS$_DEVACTIVE, and SS$_POWERFAIL.

4.4.5.2 Start Data Transfer

The start data transfer function specifies a command table that holds the parameters required to start the DR32. In addition to several other parameters, the command table contains the size and address of the command and buffer blocks, and the address of a command packet AST routine. No user privilege is required. The VAX/VMS operating system defines a single function code:

- IO$_STARTDATA—start data transfer

The start data transfer function takes one function modifier:

- IO$M_SETEVF—set event flag

If IO$M_SETEVF is included with the function code, the specified event flag is set when a command packet interrupt occurs and when the start data transfer QIO is completed. If IO$M_SETEVF is not specified, the event flag is set only when the QIO is completed.

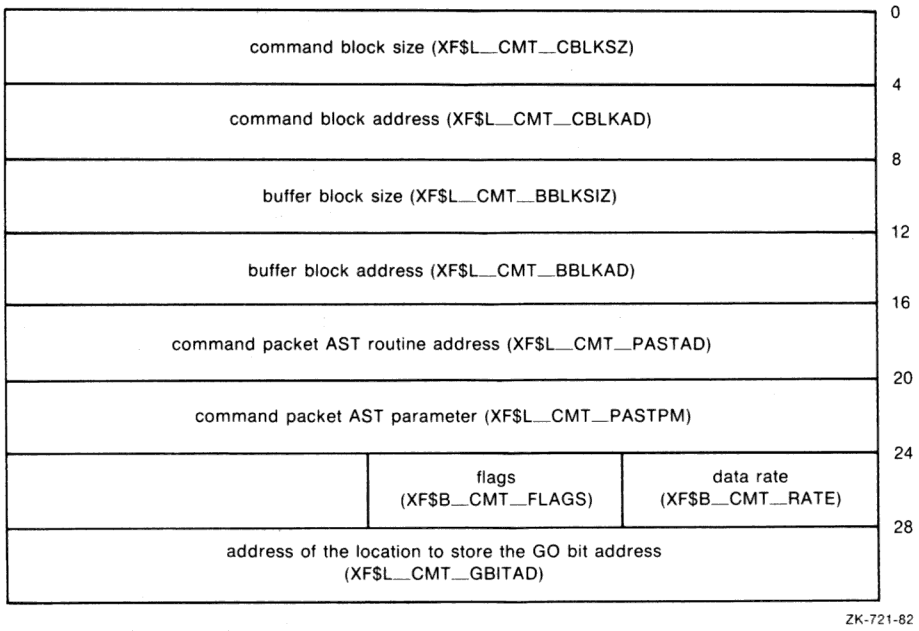
IO$M_SETEVF should not be used with the $QIOW macro, because the $QIOW will return after the event flag is set the first time.

The start data transfer function takes two device/function-dependent arguments:

- P1—the starting virtual address of the data transfer command table in the user's process
- P2—the length in bytes (always 32) of the data transfer command table (The symbolic name is XF$K_CMT_LENGTH.)

The format of the data transfer command table is shown in Figure 4–5 (offsets are shown in parentheses).

Figure 4–5 Data Transfer Command Table



Because the command block contains the queue headers for INPTQ, TERMQ, and FREEQ, its address in the second longword must be quadword-aligned.

The command packet AST routine specified in the fifth longword is called whenever the DR32 signals a command packet interrupt. A command packet AST should be distinguished from a QIO AST (**astadrs** argument). A command packet interrupt occurs whenever the DR32 completes a function and returns a packet that specifies an interrupt (see Section 4.4.3.6) by inserting it onto TERMQ. The **astadrs** argument address is called when the QIO is completed. If either the command packet AST address or the **astadrs** address is zero, the respective AST is not delivered. If the command packet specifies the set halt function, a command packet interrupt occurs regardless of the state of the packet interrupt bits.

The seventh longword contains the data rate byte and a flags byte. The data rate byte controls the DR32 clock rate. The data rate value is considered to be an unsigned integer.

For the DR780, the relationship between the value of the data rate byte and the actual data rate is given by the following formula:

$$\text{Data rate (in megabytes/sec)} = \frac{40}{(256 - \text{value of data rate byte})}$$

For example, a data rate value of 236 corresponds to an actual data rate of 2.0 megabytes/second.

For the DR750:

$$\text{Data rate (in megabytes/sec)} = \frac{12.50}{(256 - \text{value of data rate byte})}$$

For example, a data rate value of 236 corresponds to an actual data rate of .625 megabytes/second.

The maximum data rate byte values are 250 megabytes/second for the DR780 and 252 megabytes/second for the DR750.

The parameter XFMAXRATE set during system generation limits the maximum data rate that can be set. This parameter limits the maximum data rate because very high data rates on certain configurations can cause a processor timeout. If the user attempts to set the data rate higher than the rate allowed by XFMAXRATE, the error status SS$_BADPARAM is returned in the I/O status block.

The VAX/VMS operating system defines the following flag bit values:

XF$V_CMT_SETRTE	If set, XF$B_CMT_RATE specifies the data rate. If clear, the data rate established by a previous $IO_STARTDATA function is used. The IO$_LOADMCODE function sets the data rate to the last value used. If the data rate has not been previously set, a value of 0 is used.
XF$V_CMT_DIPEAB	If set, parity errors on the data portion of the DDI do not cause device aborts. If clear, a parity error results in a device abort.

The eighth longword contains the address of a location to store the address of the GO bit. This bit must be set whenever the application program inserts a command packet onto an empty INPTQ. The GO bit register is mapped in system memory space and the address is returned to the user.

The IO$_STARTDATA function locks the command and buffer blocks into memory and starts the DR32. Whenever the DR32 interrupts with a command packet interrupt, the driver queues a packet AST (if an AST address is specified) and, if IO$M_SETEVF is specified, sets the event flag. The QIO remains active until one of the following events occur:

- A set halt command packet is processed by the DR32.
- The data transfer aborts.
- A Cancel I/O on Channel ($CANCEL) system service is issued on this channel.

If an abort occurs, the second longword of the I/O status block contains additional bits that identify the cause of the abort (see Section 4.5).

The start data transfer function can return the following twelve error codes in the I/O status block:

SS$_ABORT	SS$_BUFNOTALIGN	SS$_CANCEL
SS$_CTRLERR	SS$_DEVREQERR	SS$_EXQUOTA
SS$_INSFMEM	SS$_IVBUFLEN	SS$_MCNOTVALID
SS$_NORMAL	SS$_PARITY	SS$_POWERFAIL

4.4.6 High-Level Language Interface

The VAX/VMS operating system supports a set of program-callable procedures that provide access to the DR32. The formats of these calls are given here for VAX FORTRAN users; VAX MACRO users must set up a standard VAX/VMS argument block and issue the standard procedure CALL. (Optionally, VAX MACRO users can access the DR32 directly by issuing a IO$_STARTDATA function, building command packets, and inserting them onto INPTQ.) Users of other high-level languages can also specify the proper subroutine or procedure invocation.

Six high-level language procedures are provided by the VAX/VMS operating system for the DR32. They are contained in the default system library, STARLET.OLB. Table 4–4 lists these procedures. Procedure arguments are either input or output arguments, that is, arguments supplied by the user or arguments that will contain information stored by the procedure. Except for those that are indicated as output arguments, all arguments in the following call descriptions are input arguments. By default, all procedure arguments are integer variables unless otherwise indicated.

The VAX/VMS high-level language support routines for the DR32 do the following:

- Issue I/O requests
- Allocate and manage the command memory
- Build command packets, insert them onto INPTQ, and set the GO bit
- Remove command packets from TERMQ and return the information they contain to the controlling process
- Use action routines for program–DR32 synchronization

Table 4–4 VAX/VMS Procedures for the DR32

Subroutine	Function
XF$SETUP	Defines command and buffer areas and initializes queues
XF$STARTDEV	Issues an I/O request that starts the DR32
XF$FREESET	Releases command packets onto FREEQ
XF$PKTBLD	Builds command packets and releases them onto INPTQ
XF$GETPKT	Removes a command packet from TERMQ
XF$CLEANUP	Deassigns the device channel and deallocates the command area

The VAX/VMS operating system also provides a FORTRAN parameter file, SYS$LIBRARY:XFDEF.FOR, that can be included in FORTRAN programs. This file defines many of (but not all) the XF$ . . . symbolic names described in this chapter. For example, SYS$LIBRARY:XFDEF.FOR contains symbolic definitions for function codes (that is, device control codes), interrupt control codes, command control codes, and masks for error bits set in the I/O status block and the DR32 status longword. To include these definitions in a FORTRAN program, insert the following statement in the source code:

```
INCLUDE 'SYS$LIBRARY:XFDEF.FOR'
```

4.4.6.1 XF$SETUP

The XF$SETUP subroutine defines memory space for the command and buffer areas, and initializes INPTQ, TERMQ, and FREEQ. The call to XF$SETUP must be made prior to any calls to other DR32 support routines.

The format of the XF$SETUP call is as follows:

```
CALL XF$SETUP(contxt,barray,bufsiz,numbuf,[idevmsg],
              [idevsiz],[ilogmsg],[ilogsiz],[cmdsiz],
              [status])
```

Argument descriptions are as follows:

contxt A 30-longword user-supplied array that is maintained by the support routines and is used to contain context and status information concerning the current data transfer (see Section 4.4.6.5). The **contxt** array provides a common storage area that all support routines share. For increased performance, **contxt** should be longword-aligned.

barray Specifies the starting virtual address of an array of buffers that, in the case of an output operation, contain information for transfer by the DR32, or in the case of an input operation, will contain information transferred by the DR32. For example, if **barray** is declared INTEGER*2 BARRAY (I,J), I is the size of each data buffer in words and J is the number of buffers. The lower bound on both indexes is assumed to be 1. All buffers in the array must be contiguous to each other and of fixed size.

bufsiz Specifies the size in bytes of each buffer in the array. All buffers are the same size. If the **barray** argument is declared as stated in the preceding paragraph, **bufsiz** = I*2. The **bufsiz** argument length is one longword.

numbuf Specifies the number of buffers in the array. If the **barray** argument is declared as in the preceding paragraph, **numbuf** = J. The area of memory described by the **barray**, **bufsiz**, and **numbuf** arguments is used as the buffer block for DR32 data transfers. The **numbuf** argument length is one longword.

idevmsg Specifies an array, declared by the application program, that is used to store an unsolicited input device message from the far-end DR-device. The DR32 stores unsolicited input in the device message field of a command packet from FREEQ and places that packet onto TERMQ. When XF$GETPKT removes such a packet from TERMQ, it copies the device message field into the **idevmsg** array. The calling program is then notified that information has been stored in the **idevmsg** array. The **idevmsg** argument is optional; the argument must be given if any unsolicited input is anticipated.

idevsiz Specifies the size in bytes of the **idevmsg** array. The maximum size of a device message is 256 bytes. The **idevsiz** argument is optional; if **idevmsg** is specified, **idevsiz** must be specified. The **idevsiz** argument length is one word.

ilogmsg Specifies an array, declared by the application program, that is used to store log information from the far-end DR-device contained in the log area field of the command packet. Log information is hardware-dependent data that is returned by the far-end DR-device. The XF$SETUP routine stores the address and size of the **ilogmsg** array; the log information is stored in the **ilogmsg** array by the XF$GETPKT routine. The **ilogmsg** argument is optional; the argument must be given if any log information is anticipated.

ilogsiz Specifies the size in bytes of the **ilogmsg** array. The maximum size of a log message is 256 bytes. The **ilogsiz** argument is optional. However, if **ilogmsg** is specified, **ilogsiz** must be specified. The **ilogsiz** argument length is one word.

cmdsiz Specifies the amount of memory space to be allocated from which command packets are to be built. The user must consider the following factors when deciding how much memory to allocate for this purpose:

1. The number of command packets that the application program will be using
2. That the device message and log area fields in command packets are rounded up to quadword boundaries
3. That the size of the command packet itself is rounded up to an eight-byte boundary
4. That **cmdsiz** will be rounded up to a page boundary

The **cmdsiz** argument is optional; argument length is one longword. If defaulted, the allocated space is equal to the following, which is rounded up to a full page.

```
(numbuf)*(32+idevsiz+ilogsiz)*(3)
```

Memory space for command packets is obtained by calling LIB$GET_VM.

status This output argument receives the VAX/VMS success or failure code of the XF$SETUP call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Invalid input argument

Error returns can be found in LIB$GET_VM.

The **status** argument is optional; argument length is one longword.

4.4.6.2 XF$STARTDEV

The XF$STARTDEV subroutine issues the I/O request that starts the DR32 data transfer.

The format of the XF$STARTDEV call is as follows:

```
CALL XF$STARTDEV(contxt,devnam,[pktast],[astparm],[efn],-
        [modes],[datart],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

devnam Specifies the device name (logical name or actual device name) of the DR32. All letters in the resultant string must be capitalized and the device name must terminate with a colon, for example, XFA0:. The **devnam** datatype is character string.

pktast Specifies the address of an AST routine that is called each time a command packet that specifies an interrupt in its interrupt control field is returned by the DR32, that is, placed onto TERMQ (see Section 4.4.7.3). This AST routine is also called on completion of the I/O request. Normally, the AST routine would call XF$GETPKT to remove command packets from TERMQ until TERMQ is empty. The **pktast** argument is optional.

astparm Specifies a longword parameter that is included in the call to the **pktast**-specified AST routine. The format used to call the AST routine is:

```
CALL pktast(astparm)
```

The **astparm** argument is optional; argument length is one longword. If **astparm** is not specified, **pktast** is called with no parameter.

efn If the event flag must be determined by the application program, **efn** specifies the number of the event flag that is set when a packet interrupt is delivered. Otherwise, it is not necessary to include this argument in a XF$STARTDEV call. If defaulted, **efn** is 21. The **efn** argument length is one word.

The event flag (either the default or the event flag specified by this argument) is set for every packet interrupt, and also when the QIO completes.

modes Specifies the mode of operation. The VAX/VMS operating system defines the following value:

2 = parity errors on the data portion of the DDI that do not cause the device to abort.

If defaulted, **modes** is 0 (a parity error causes the device to abort).

datart Specifies the data rate. The data rate controls the speed at which the transfer takes place. The data rate is considered to be an unsigned integer in the range 0 to 255. The relationship between the specified data rate value and the actual data rate for the DR780 is given by the following formula:

$$\text{Data rate (in megabytes/sec)} = \frac{40}{(256 - \text{value of data rate byte})}$$

For example, a data rate value of 236 corresponds to an actual data rate of 2.0 megabytes/second. Note that the DR780 ignores data rate values greater than 250.

(See Section 4.4.5.2 for the DR750 formula.)

If **datart** is defaulted, the previously set data rate is used. The **datart** argument length is one byte.

status This output argument receives the VAX/VMS success or failure code of the XF$STARTDEV call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Required parameter defaulted

Error returns can be obtained by issuing the Create I/O on Channel ($CREATE) and Queue I/O Request ($QIO) system services.

The **status** argument is optional; argument length is one longword.

4.4.6.3 XF$FREESET

The XF$FREESET subroutine releases command packets onto FREEQ. These packets are then available to the DR780 to store any unsolicited input from the far-end DR-device. If unsolicited input from the far-end DR-device is expected, the XF$FREESET call should be made before the XF$STARTDEV call is issued.

Idevsiz the argument that specifies the size of the **idevmsg** array in the call to XF$SETUP, defines the size of the device message field in command packets inserted onto FREEQ. This occurs because unsolicited device messages are copied from the device message field of the command packet to the **idevmsg** array.

Note that the XF$FREESET subroutine may occasionally disable ASTs for a very short period.

The format of the XF$FREESET call is as follows:

```
CALL XF$FREESET(contxt,[numpkt],[intctrl],[action],-
                [actparm],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

numpkt Specifies the number of command packets to be released onto FREEQ. The **numpkt** argument is optional; argument length is one word. If defaulted, **numpkt** is 1.

intctrl Specifies the conditions under which an AST is delivered (and the event flag set) when the DR32 places this command packet (or packets) on TERMQ (see Section 4.4.6.2). The VAX/VMS operating system defines the following values:

```
0 = unconditional AST delivery and event
    flag set
1 = AST delivery and event flag set only
    if TERMQ is empty
2 = no AST interrupt or event flag set
```

The **intctrl** argument is optional; argument length is one word. If defaulted, **intctrl** is 0.

action Specifies the address of a routine that is called when any command packet built by this call to XF$FREESET is removed from TERMQ by XF$GETPKT (see Section 4.4.7.3). The **action** argument is optional.

actparm A longword parameter that is passed to the action routine when the action routine is called (see Section 4.4.7.3). The **actparm** argument is optional.

status This output argument receives the VAX/VMS success or failure code of the XF$FREESET call:

SS$_NORMAL	Normal successful completion
SS$_BADQUEUEHDR	FREEQ interlock timeout
SS$_INSFMEM	Insufficient memory to build command packets
SHR$_NOCMDMEM	Command memory not allocated (usually because the data transfer has stopped and XF$CLEANUP has been called, or because XF$SETUP has not been called)

4.4.6.4 XF$PKTBLD

The XF$PKTBLD subroutine builds command packets and releases them onto INPTQ.

Note that the XF$PKTBLD subroutine may occasionally disable ASTs for a very short period.

The format of the XF$PKTBLD call is as follows:

```
CALL XF$PKTBLD(contxt,func,[index],[size],
        [devmsg],[devsiz],[logsiz],[modes],
        [action],[actparm],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

func Specifies the device control code. Device control codes describe the function the DR32 is to perform. The **func** argument length is one word. The VAX/VMS operating system defines the following values (Table 4–2 describes the functions in greater detail):

Symbol	Value	Function
XF$K_PKT_RD	0	Read device
XF$K_PKT_RDCHN	1	Read device chained
XF$K_PKT_WRT	2	Write device
XF$K_PKT_WRTCHN	3	Write device chained
XF$K_PKT_WRTCM	4	Write device control message
	5	(reserved)
XF$K_PKT_SETTST	6	Set self-test
XF$K_PKT_CLRTST	7	Clear self-test
XF$K_PKT_NOP	8	No operation
XF$K_PKT_DIAGRI	9	Diagnostic read internal
XF$K_PKT_DIAGWI	10	Diagnostic write internal
XF$K_PKT_DIAGRD	11	Diagnostic read DDI
XF$K_PKT_DIAGWC	12	Diagnostic write control message
XF$K_PKT_SETRND	13	Set random enable
XF$K_PKT_CLRRND	14	Clear random enable
XF$K_PKT_HALT	15	Set halt

index Specifies the index of a data buffer specified by the **barray** argument (see Section 4.4.6.1). The specific index value given means that elements **barray** (1,index) through **barray** (size,index) will be transferred, that is, one buffer full of data. The **index** argument is optional and is only used when the function specifies a data transfer, that is, a read device, read device chained, write device, or write device chained function. The **index** argument length is one word.

size Specifies a byte count to be transferred. This argument is optional and is only used when the function specifies a data transfer. If defaulted, the number of bytes to be transferred is assumed to be the size of the buffer (specified by the **bufsiz** argument in the call to XF$SETUP). If the **size** argument is given, then the specified number of bytes of data **barray** (1,index) through **barray** (size,index) will be transferred. If **size** is defaulted and the function specifies a data transfer, then **barray** (1,index) through **barray** (bufsiz,index) will be transferred. The **size** argument length is one longword.

devmsg Specifies a variable that contains the device message to be sent to the far-end DR-device. Provides additional control of the far-end DR-device (see Section 4.4.3.12). The **devmsg** argument is optional.

devsiz Specifies the size in bytes of the **devmsg** variable. If the **modes** argument specifies that a device message is to be sent over the control portion of the DDI, **devsiz** specifies the number of bytes of **devmsg** that will be sent to the far-end DR-device.

logsiz Specifies the size of the log message expected from the far-end DR-device. The **logsiz** argument is optional, argument length is one word. If defaulted, **logsiz** is 0.

modes Provides additional control of the transaction. The VAX/VMS operating system defines the following values:

Value	Meaning
+8	Only the function code is sent over the control portion of the DDI to the far-end DR-device. Only for read device, read device chained, write device, and write device chained functions.
+16	The function code and the device message are sent over the control portion of the DDI to the far-end DR-device. Only for read device, read device chained, write device, and write device chained functions.
+24	The function code, the device message, and the buffer size are sent over the control portion of the DDI to the far-end DR-device. Only for read device, read device chained, write device, and write device chained functions.
	If none of the preceding three values is selected, nothing is transmitted over the control portion of the DDI to the far-end DR-device.

Value	Meaning
+32	Length errors are suppressed. If not selected, a length error results in an abort.
+64	An AST should be delivered (and an event flag set) when this command packet is inserted onto TERMQ, provided TERMQ is empty.
+128	No AST is delivered or event flag set for this command packet. If both +64 and +128 are selected, +128 takes precedence. If neither of the preceding two values is selected, ASTs are delivered and the event flag is set unconditionally, that is, whenever this command packet is placed onto TERMQ.
+256	Insert this command packet at the head of INPTQ. If not selected, insert the packet at the tail of INPTQ.

The **modes** argument default value is 0.

action Specifies the address of a routine that is called when XF$GETPKT removes this command packet from TERMQ. This occurs after the DR32 has completed the command specified in the packet (see Section 4.4.7.3). The **action** argument length is one longword.

actparm A longword parameter that is passed to the action routine when the action routine is called (see Section 4.4.7.3). The **actparm** argument is optional.

status This output argument receives the VAX/VMS success or failure code of the XF$PKTBLD call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Input parameter error
SS$_BADQUEUEHDR	INPTQ interlock timeout
SS$_INSFMEM	Insufficient memory to build command packets
SHR$_NOCMDMEM	Command memory not allocated (usually because the data transfer has stopped and XF$CLEANUP has been called, or because XF$SETUP has not been called)

4.4.6.5 XF$GETPKT

The XF$GETPKT subroutine removes a command packet from TERMQ.

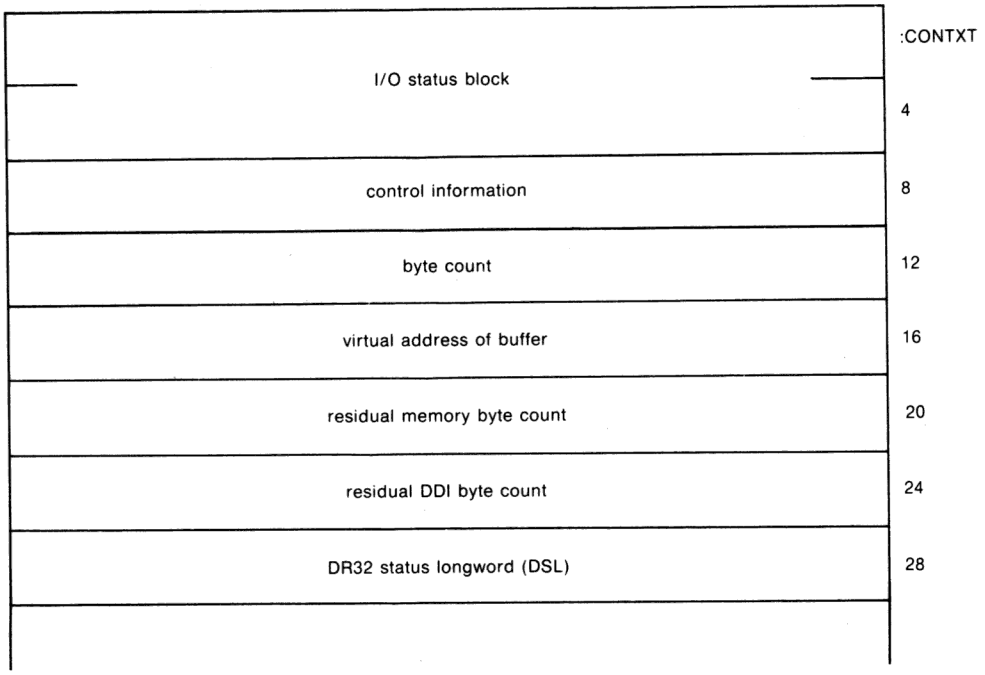
Note that the XF$GETPKT subroutine may occasionally disable ASTs for a very short period.

The format of the XF$GETPKT call is as follows:

```
CALL XF$GETPKT(contxt,[waitflg],[func],[index],-
        [devflag],[logflag],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains the context and status information (see Section 4.4.6.1). On return from XF$GETPKT, the first eight longwords of the **contxt** array are filled with the status of the data transfer:



ZK-722-82

The first two longwords are the I/O status block. The next six longwords are copied directly from bytes 8 through 31 of the command packet.

This context and status information is returned by the DR32 as status in each command packet. With the exception of the I/O status block, the information is copied by XF$GETPKT into the **contxt** array whenever XF$GETPKT removes a command packet from TERMQ.

The I/O status block is stored only after the data transfer has halted and it contains the final status of the transfer. Section 4.5 describes the I/O status block.

(See Section 4.4.2 for a description of the remaining fields.)

waitflg Specifies the consequences of an attempt by XF$GETPKT to remove a command packet from an empty TERMQ. If **waitflg** is 0 (default), XF$GETPKT waits for the event flag to be set and then removes a packet from TERMQ. If **waitflg** is 1, XF$GETPKT returns immediately with a failure status. Normally, **waitflg** is set to 1 (.TRUE.) for AST synchronization and set to 0 (.FALSE.) for event flag synchronization (see Section 4.4.7). The **waitflg** argument is optional.

func This output argument receives the device control code specified in this command packet (see Section 4.4.6.4). The **func** argument is optional; argument length is one word.

index If the current command packet specified a data transfer, this output argument receives the buffer index specified when this command packet was built by XF$PKTBLD (see Section 4.4.6.4). The **index** argument is optional; argument length is one word.

devflag If set to .TRUE. (255), this output argument indicates that a device message was stored in the **idevmsg** array, which is described in the XF$SETUP call (see Section 4.4.6.1). The **devflag** argument is optional; argument length is one byte.

logflag If set to .TRUE. (255), this output argument indicates that a log message was stored in the **ilogmsg** array, which is described in the XF$SETUP call (see Section 4.4.6.1). The **logflag** argument is optional; argument length is one byte.

status This output argument receives the status of the XF$GETPKT call:

SS$_NORMAL	Normal successful completion
SS$_BADQUEUEHDR	TERMQ interlock timeout
SHR$_QEMPTY	TERMQ empty but transfer still in progress; only returned if **waitflg** is .TRUE
SHR$_HALTED	TERMQ empty, transfer complete, and I/O status block contains final status; XF$CLEANUP called automatically (Subsequent calls to XF$GETPKT return SHR$_NOCMDMEM.)
SHR$_NOCMDMEM	Command memory not allocated; usually indicates either: **1** XF$SETUP not called **2** XF$CLEANUP called

4.4.6.6 XF$CLEANUP

The XF$CLEANUP subroutine deassigns the channel and deallocates the command area allocated by XF$SETUP. If XF$GETPKT detects a TERMQ empty condition and the transfer has halted, it will automatically call XF$CLEANUP. However, if the transfer either terminates in a SS$_CTRLERR or SS$_BADQUEHDR error, or is intentionally terminated, XF$GETPKT may not detect these conditions and XF$CLEANUP should be called explicitly.

The format of the XF$CLEANUP call is as follows:

```
CALL XF$CLEANUP(contxt,[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

status This output argument receives the status of the XF$CLEANUP call:

SS$_NORMAL	Normal successful completion
SHR$_NOCMDMEM	The command memory not allocated; there are error returns from LIB$FREE_VM and $DASSIGN

4.4.7 User Program–DR32 Synchronization

Synchronization of high-level language application programs with the DR32 can be achieved in three ways:

- Event flags
- AST routines
- Action routines

4.4.7.1 Event Flags

Event flag are synchronized by calling the XF$GETPKT routine (see Section 4.4.6.5) with the **waitflg** argument set to 0 (default). The **pktast** argument in the XF$STARTDEV routine (see Section 4.4.6.2) is normally set to its default value. If the XF$GETPKT routine is called and the termination queue is empty, the routine waits until the DR32 places a command packet on the queue and sets the event flag. The packet is then removed from the queue and returned to the caller.

4.4.7.2 AST Routines

If a call to the XF$STARTDEV routine includes the **pktast** argument, the specified AST routine is called each time an AST is delivered. AST delivery can be controlled on a packet-by-packet basis by using the **intctrl** argument in the XF$FREESET routine and by specifying appropriate values in the **modes** argument of the XF$PKTBLD routine (see Sections 4.4.6.3 and 4.4.6.4). For a particular command packet, ASTs can be delivered as follows:

- Unconditionally when the packet is placed onto TERMQ
- Only if TERMQ is empty when the packet is placed on it
- Not at all (That is, there is no AST when the packet is placed on TERMQ.)

There is no guarantee that an AST will be delivered for every command packet, even when the **astctrl** argument indicates unconditional AST delivery. In particular, if packet interrupts are closely spaced, several packets may be placed onto TERMQ even though only one AST is delivered. Therefore, the AST routine should continue to call the XF$GETPKT routine until all command packets are removed from TERMQ.

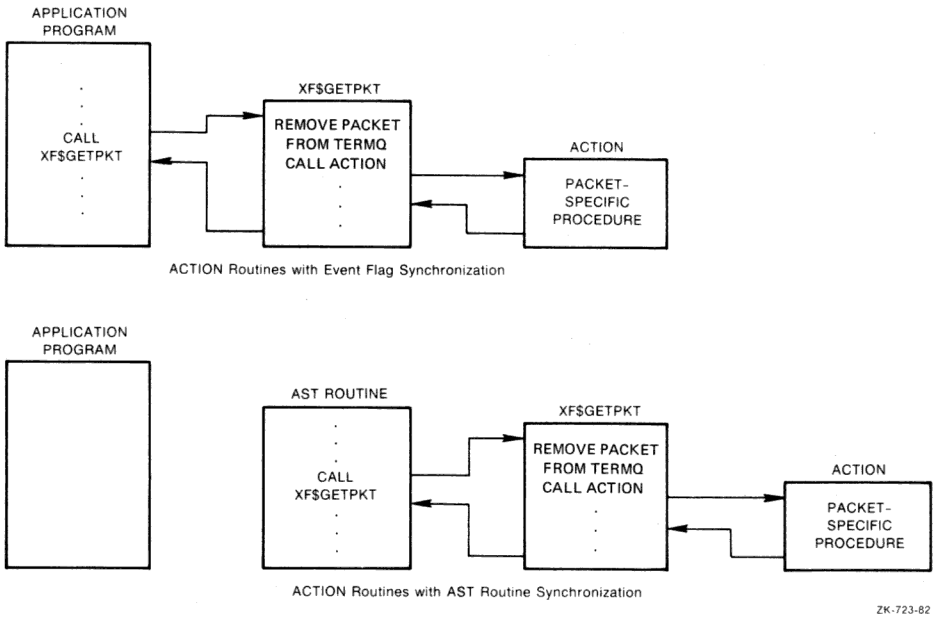
4.4.7.3 Action Routines

The **action** argument specified in the XF$FREESET and XF$PKTBLD routines (see Sections 4.4.6.3 and 4.4.6.4) can be used for a more automated synchronization of the program with the DR32. Routines specified by **action** arguments can be used for both event flag and AST routine synchronization.

The address of the action routine is included in the command packet. This routine is automatically called by the XF$GETPKT routine when it removes that packet from TERMQ. This allows the user to define, at the time it is built, how the command packet will be handled once it is removed from TERMQ. In addition to specifying different action routines for different types of command packets, the user can also specify an action routine parameter (**actparm**) to further identify the command packet or the action to be taken when the command is completed. Figure 4–6 shows the use of action-specified routines for program synchronization.

An important difference between AST routine and action routine use is the number of times the respective routines are specified. Command packet AST routines are specified only once, in a XF$STARTDEV call; a single AST routine is implied. Action routines, however, are specified in each command packet. This allows a different action routine to be designed for each type of command packet.

Figure 4–6 Action Routine Synchronization



Routines specified by the action argument are supplied by the user. The format of the calling interface is as follows:

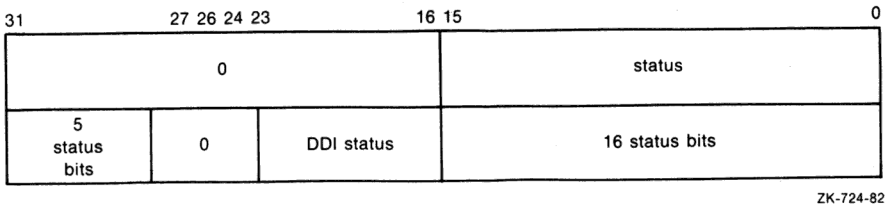
```
CALL action-routine (contxt,actparm,devflag,logflag,
                     func,index,status)
```

With the exception of **actparm**, all arguments are the same as those described for the XF$GETPKT routine. In effect, the action routine will receive the same information XF$GETPKT optionally returns to its calling program, along with the **actparm** argument that was specified when the packet was built. If these variables are to be passed as inputs to the action routine, they must be supplied as output variables in the call to the XF$GETPKT routine.

4.5 I/O Status Block

The I/O status block for the load microcode and start data transfer QIO functions is shown in Figure 4–7. The I/O status block used in the first two longwords of the **contxt** array for high-level language calls also has the same format.

Figure 4–7 I/O Functions IOSB Contents



VAX/VMS status values are returned in the first longword. Appendix A lists these values. (The *VAX/VMS System Messages and Recovery Procedures Reference Manual* provides explanations and user actions for these returns.) If SS$_CTRLERR, SS$_DEVREQERR, or SS$_PARITY is returned in the status word, the second longword contains additional returns, that is, device-dependent data. Table 4–5 lists these returns.

The I/O status block for an I/O function is returned after the function completes. Status is not stored on the completion of every command packet, because any number of packets can pass between the application program and the DR32 when a single QIO executes.

Table 4–5 Device-Dependent IOSB Returns for I/O Functions

Symbolic Name	Meaning
	16 Status Bits
XF$V_PKT_SUCCESS	The command was performed successfully.
XF$V_IOS_CMDSTD	The command specified in the command packet started.
XF$V_IOS_INVPTE	An invalid page table entry.
XF$V_IOS_FREQPK	This command packet came from FREEQ.
XF$V_IOS_DDIDIS	The far-end DR-device is disabled.
XF$V_IOS_SLFTST	The DR32 is in self-test mode.
XF$V_IOS_RNGERR	The user-provided address is outside the command block range or the buffer block range.
XF$V_IOS_UNQERR	A queue element was not aligned on a quadword boundary.
XF$V_IOS_INVPKT	A packet was not a valid DR32 command packet.
XF$V_IOS_FREQMT	A message was received from the far-end DR-device and FREEQ was empty.
XF$V_IOS_RNDENB	Random access mode is enabled.
XF$V_IOS_INVDDI	A protocol error occurred on the DDI.
XF$V_IOS_LENERR	The far-end DR-device terminated the data transfer before the required number of bytes was sent, or a message was received from the far-end DR-device and the device message field in the command packet at the head of FREEQ was not large enough to hold it.
XF$V_IOS_DRVABT	The I/O driver aborted the DR32 function.
XF$V_PKT_PARERR	A parity error occurred on the data or control portion of the DDI.

Table 4–5 (Cont.) Device-Dependent IOSB Returns for I/O Functions

Symbolic Name	Meaning
	DDI Status
XF$V_IOS_DDISTS	The one-byte status register 0 for the far-end DR-device. XFV_IOS_NEXREG, XFV_IOS_LOG, and XF$V_IOS_DDIERR are returns from this register.
XF$V_IOS_NEXREG	An attempt was made to access a nonexistent register on the far-end DR-device.
XF$V_IOS_LOG	The far-end DR-device registers are stored in the log area.
XF$V_IOS_DDIERR	An error occurred on the far-end DR-device.
	5 Status Bits
XF$V_IOS_BUSERR	An error on the processor's internal CPU memory bus occurred.
XF$V_IOS_RDSERR	A noncorrectable memory error occurred (read) data substitute.
XF$V_IOS_WCSPE	Writeable control store (WCS) parity error.
XF$V_IOS_CIPE	Control interconnect parity error. A parity error occurred on the control portion of the DDI.
XF$V_IOS_DIPE	Data interconnect parity error. A parity error occurred on the data portion of the DDI.

4.6 Programming Hints

This section contains information on important programming considerations relevant to users of the DR32 driver.

4.6.1 Command Packet Prefetch

The DR32 has the capability of prefetching command packets from INPTQ. While executing the command specified in one packet, the DR32 can prefetch the next packet, decode it, and be ready to execute the specified command at the first opportunity. When the command is executed depends on which command is specified. For example, if two read device or write device command packets are on INPTQ, the DR32 fetches the first packet, decodes the command, verifies that the transfer is legal, and starts the data transfer. While the transfer is taking place, the DR32 prefetches the next read device or write device command packet, decodes it, and verifies the transfer legality. The second transfer begins as soon as the first transfer is completed.

On the other hand, if the two command packets on INPTQ are read device (or write device) and write device control message, in that order, the DR32 prefetches the second packet and immediately executes the command, because control messages can be overlapped with data transfers. The DR32 then prefetches the next command packet. In an extreme case, the DR32 can send several control messages over the control portion of the DDI while a single data transfer takes place on the data portion of the DDI.

The prefetch capability and the overlapping of control and data transfers can cause unexpected results when programming the DR32. For instance, if the application program calls for a data transfer to the far-end DR-device followed by notification of the far-end DR-device that data is present, the program cannot simply insert a write device command packet and then a write control message command packet onto INPTQ—the control message may very likely arrive before the data transfer completes.

A better way to synchronize the data transfer with notification of data arrival is to request an interrupt in the interrupt control field of the data transfer command packet. Then, when the data transfer command packet is removed from TERMQ, the application program can insert a write control message command packet onto INPTQ to notify the far-end DR-device that the data transfer has completed.

Another consequence of command packet prefetching occurs when, for example, two write device command packets are inserted onto INPTQ. While the first data transfer takes place, the second command packet is prefetched and decoded. If an unusual event occurs and the application program must send an immediate control message to the far-end DR-device, the application program may insert a write device control message packet onto INPTQ. However, this packet is not sent immediately because the second write device command packet has already been prefetched; the control message is sent after the second data transfer starts.

If the application program must send a control message with minimum delay, use one of the following techniques:

- Insert only one data transfer function onto INPTQ at a time. If this is done, a second transfer function will not be prefetched and a control message can be sent at any time.
- Use smaller buffers or a faster data rate to reduce the time necessary to complete a given command packet.
- Issue a Cancel I/O on Channel ($CANCEL) system service call followed by another IO$_STARTDATA function.

4.6.2 Action Routines

Action routines provide a useful DR32 programming technique. They can be used in application programs written in either assembly language or a high-level language. When a command packet is built, the address of a routine to be executed when the packet is removed from TERMQ is appended to the end of the packet. Then, rather than having to determine what action to perform for a particular packet when it is removed from TERMQ, the specified action routine is called.

4.6.3 Error Checking

Bits 0 through 23 in the second longword of the I/O status block correspond to the same bits in the DR32 status longword (DSL). Although the I/O status block is written only after the QIO function completes, the DSL is stored in every command packet. However, because there is no command packet in which to store a DSL for certain error conditions, for example, FREEQ empty, some errors are reported only in the I/O status block. To check for an error under these conditions, the user should examine the DSL in each packet for success or failure only. Then, if a failure occurs, the specific error can be determined from the I/O status block. The I/O status block should also be checked to verify that the QIO has not completed prior to a wait for the insertion of additional command packets onto TERMQ. In this way, the application program can detect asynchronous errors for which there is no command packet available.

4.6.4 Queue Retry Macro

When an interlocked queue instruction is included in the application program, the code should perform a retry if the queue is locked. However, the code should not execute an indefinite number of retries. Consequently, all retry loops should contain a maximum retry count. The macro programming example provided in Section 4.7 contains a convenient queue retry macro.

4.6.5 Diagnostic Functions

The diagnostic functions listed in Table 4–2 can be used to test the DR32 without the presence of a far-end DR-device. For the DR780, the user should perform the following test sequence:

1. Insert a set self-test command packet onto INPTQ.
2. Insert a diagnostic write internal command packet that specifies a 128-byte buffer onto INPTQ. This packet copies 128 bytes from memory into the DR780 internal data silo.
3. Insert a diagnostic read DDI command packet onto INPTQ. This packet transmits the 128 bytes of data from the silo over the DDI and returns it to the silo.
4. Insert a diagnostic read internal command packet that specifies another 128-byte buffer in memory onto INPTQ. This packet copies 128 bytes of data from the silo into memory.
5. Compare the two memory buffers for equality. Note that on the DR780, the diagnostic read internal function destroys the first four bytes in the silo before storing the data in memory. Therefore, compare only the last 124 bytes of the two buffers.
6. Insert a clear self-test command packet onto INPTQ.

4.6.6 The NOP Command Packet

It is often useful to insert a NOP command packet onto INPTQ to test the state of the DDI disable bit (XF$M_PKT_DDIDIS in the DSL). By checking this bit before initiating a data transfer, an application program can determine whether the far-end DR-device is ready to accept data.

4.6.7 Interrupt Control Field

As described in Section 4.4.3.6, the interrupt control field determines the conditions under which an interrupt is generated: unconditionally, if TERMQ was empty, or never. There are several general applications of this field:

- If a program performs five data transfers and requires notification of completion only after all five have completed, the first four command packets should specify no interrupt and the fifth command packet should specify an unconditional interrupt.
- If a program performs a continuous series of data transfers, for example, each command packet can specify interrupt only if TERMQ was empty. Then, every time an event flag or AST notifies the program that a command packet was inserted onto TERMQ, the program removes and processes all packets on TERMQ until it is empty.
- Command packets that specify no interrupt should never be mixed with command packets that specify an interrupt if TERMQ was empty. If this were done, a command packet that specifies no interrupt could be inserted onto TERMQ followed by a command packet that specifies interrupt if TERMQ was empty. Then the latter packet would not interrupt and the program would never be notified that command packets were inserted onto TERMQ.

4.7 Programming Examples

The program examples in the following two sections use DR32 high-level language procedures and DR32 Queue I/O functions.

4.7.1 DR32 High-Level Language Program

This program (Example 4–1) is an example of how the DR32 high-level language procedures perform a data transfer from a far-end DR-device. The program reads a specified number of data buffers from an undefined far-end DR-device, which is assumed to be a data source, into the VAX memory. The number of buffers is controlled by the NUMBUF parameter. The program contains examples of the read data chained function code and DR32 application program synchronization using AST routines and action routines.

Example 4–1 DR32 High-Level Language Program Example

```
************************************************************************
C
C                DR32 HIGH-LEVEL LANGUAGE PROGRAM
C
************************************************************************
       INCLUDE 'XFDEF.FOR'                      ;DEFINE XF CONSTANTS
       PARAMETER       BUFSIZ = 1024            !SIZE OF EACH BUFFER
       PARAMETER       NUMBUF = 8               !NUMBER OF BUFFERS IN
                                                !RING
       PARAMETER       ILOGSIZ = 4              !SIZE OF INPUT LOG
                                                !ARRAY
       PARAMETER       EFN = 0                  !EVENT FLAG SYNCHRON-
                                                !IZING MAIN LEVEL WITH
                                                !AST ROUTINE
       INTEGER*2       BUFARRAY(BUFSIZ,NUMBUF) !THE RING OF BUFFERS
       INTEGER*2       INDEX                    !REFERS TO BUFFER
                                                !IN BUFARRAY
       INTEGER*2       COUNT                    !COUNTS NUMBER OF
                                                !BUFFERS FILLED
       INTEGER*2       DATART                   !DR32 CLOCK RATE
       INTEGER*4       CONTXT(30)       !CONTEXT ARRAY USED BY SUPPORT
       INTEGER*4       ILOGMSG(ILOGSIZ)!LOG MESSAGES FROM DEVICE
                                        !STORED HERE
       INTEGER*4       STATUS           !RETURNS FROM SUBROUTINES
       INTEGER*4       DEVMSG           !far-end DR-DEVICE CODE
       EXTERNAL        ASTRTN           !AST ROUTINE
       EXTERNAL        AST$PROCBUF      !ACTION ROUTINE TO HANDLE
                                        !COMPLETION OF READ DATA
                                        !COMMAND PACKET
       EXTERNAL        AST$HALT         !ACTION ROUTINE TO HANDLE
                                        !COMPLETION OF A HALT
                                        !COMMAND PACKET
       COMMON  /MAIN_AST/       CONTXT, INDEX
       COMMON  /MAIN_ACTION/    BUFARRAY, ILOGMSG, COUNT
       EXTERNAL        SS$_NORMAL       !SUCCESS STATUS RETURN
************************************************************************
C
C  THE CALL TO THE SETUP ROUTINE
C
************************************************************************
       CALL XF$SETUP (CONTXT,BUFARRAY,BUFSIZ*2,NUMBUF,,,ILOGMSG,
       1              ILOGSIZ*4,,STATUS)
       IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))
C
C  PRELOAD THE INPUT QUEUE BEFORE STARTING THE DR32 IN ORDER TO AVOID
C  A DELAY IN THE DATA TRANSFER
C
C
```

(Continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
************************************************************************
C
C  BUILD COMMAND PACKETS
C
************************************************************************
C  BUILD THE COMMAND PACKET THAT WILL INSTRUCT THE far-end DR-DEVICE
C  TO START SAMPLING.  ARBITRARILY ASSUME THAT THE far-end DR-DEVICE
C  WILL RECOGNIZE THIS DEVICE MESSAGE.  INSERT THIS PACKET ON THE
C  INPUT QUEUE (INPTQ).
C
       DEVMSG = 25                      !SIGNAL far-end DR-DEVICE
                                        !"GO"

       CALL XF$PKTBLD (
       1       CONTXT,                  !THE CONTEXT ARRAY
       1       XF$K_PKT_WRTCM,          !WRITE CONTROL MESSAGE
                                        !FUNCTION
       1       ,,                       !NO INDEX OR SIZE
       1       DEVMSG,                  !SIGNAL "GO"
       1       4,                       !SIZE OF DEVMSG IN BYTES
       1       ILOGSIZ*4                !SPACE FOR INPUT LOG
                                        !MESSAGE
       1       XF$K_PKT_UNCOND          !MODES: UNCONDITIONAL
                                        !       INTERRUPT
       1       + XF$K_PKT_CBDM          !     : SEND FUNC AND DEVMSG
       1       + XF$K_PKT_INSTL         !     : INSERT PACKET AT INPTQ
                                        !       TAIL
       1       ''                       !NO ACTION ROUTINE OR ACTPARM
       1       STATUS)
       IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))
C
C  IN A LOOP, BUILD THE COMMAND PACKETS THAT WILL PERFORM THE CHAINED
C  READ TO INITIALLY FILL THE BUFFERS
C
       DO 10   INDEX = 1, NUMBUF        !FOR ALL BUFFERS DO
               CALL XF$PKTBLD(
       1       CONTXT,                  !THE CONTEXT ARRAY
       1       XF$K_PKT_RDCHN,          !READ DATA CHAINED
       1       INDEX,                   !IDENTIFIES BUFFER
       1       ,,,                      !NO SIZE, DEVMSG, OR DEVSIZ
       1       ILOGSIZ*4,               !SPACE FOR INPUT LOG MESSAGE
       1       XF$K_PKT_UNCOND          !MODES: UNCONDITIONAL
                                        !       INTERRUPT
       1       + XF$K_PKT_CB            !     : SEND FUNCTION CODE
       1       + XF$K_PKT_INSTL,        !     : INSERT PACKET AT INPTQ
                                        !       TAIL
       1       AST$PROCBUF,             !ACTION ROUTINE
       1       ,                        !NO ACTPARM
       1       STATUS)
       IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))
10     CONTINUE
C
C  THE INPUT QUEUE IS LOADED
C
```

(Continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
************************************************************************
C
C  START THE DR32
C
************************************************************************
        DATART = 0                      !DATA TRANSFER RATE
        COUNT = 0                       !NUMBER OF BUFFERS THAT HAVE
                                        !BEEN FILLED
        CALL SYS$CLREF (%VAL(EFN))      !CLEAR EVENT FLAG BEFORE START
        CALL XF$STARTDEV (CONTXT,'XFAO:',ASTRTN,,,,DATART,STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))
C
C  FROM THIS POINT, ROUTINES AT THE AST LEVEL ASSUME CONTROL.  WAIT
C  FOR THEM TO SIGNAL COMPLETION OF THE SAMPLING SWEEP.
C
        CALL SYS$WAITFR (%VAL(EFN))
        STOP
        END
************************************************************************
C
C  AST ROUTINES
C
************************************************************************
        SUBROUTINE      ASTRTN (ASTPARM)
        INCLUDE 'XFDEF.FOR/NOLIST'
        INTEGER*2       ASTPARM                 !UNUSED PARAMETER
        INTEGER*4       CONTXT(30)              !CONTEXT ARRAY
        INTEGER*4       STATUS                  !FOR CALL TO XF$GETPKT
        LOGICAL*1       WAITFLG                 !INPUT TO XF$GETPKT
        LOGICAL*1       LOGFLAG                 !INPUT TO XF$GETPKT
        COMMON  /MAIN_AST/      CONTXT, INDEX
        EXTERNAL        SS$_NORMAL
C
C  CALL XF$GETPKT IN A LOOP UNTIL TERMQ IS EMPTY.  XF$GETPKT WILL CALL
C  THE APPROPRIATE ACTION ROUTINE FOR EACH COMMAND PACKET.
C
        WAITFLG = .TRUE.                !DO NOT WAIT FOR EVENT FLAG
        LOGFLAG = .TRUE.                !REQUEST NOTIFICATION IF LOG
                                        !MESSAGE IS IN PACKET
10      CALL XF$GETPKT (CONTXT,WAITFLG,,INDEX,,LOGFLAG,STATUS)
        IF (STATUS .EQ. %LOC(SS$_NORMAL))       !PACKET FROM TERMQ
        1       GOTO 10
        IF (STATUS .EQ. SHR$_QEMPTY)    !TERMQ EMPTY - TRANSFER
        1       GOTO 20                 !STILL IN PROGRESS
        IF (STATUS .EQ. SHR$_HALTED .OR. STATUS .EQ. SHR$_NOCMDMEM)
        1       GOTO 20                 !TRANSFER COMPLETE.  NO MORE
                                        !COMMAND PACKETS. ASTS MAY
                                        !STILL BE DELIVERED
        CALL LIB$STOP (%VAL(STATUS))    !ERROR IN XF$GETPKT
20      RETURN
        END
```

(Continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
************************************************************************
C
C  ACTION ROUTINE
C
************************************************************************
        SUBROUTINE      AST$PROCBUF (CONTXT,ACTPARM,DEVFLAG,LOGFLAG,
        1                            FUNC,INDEX,STATUS)
C
C  THIS IS THE ACTION ROUTINE CALLED BY XF$GETPKT WHEN IT REMOVES A
C  COMMAND PACKET FROM TERMQ.  THIS PACKET HAS JUST COMPLETED A READ
C  DATA OPERATION FROM THE BUFFER SPECIFIED BY INDEX.  THE BUFFER IS
C  PROCESSED, AND IF MORE DATA IS REQUIRED, THAT IS, BUFCOUNT .LE.
C  MAXCOUNT), ANOTHER PACKET IS BUILT.  THE BUFFER IN THIS PACKET IS
C  THEN REFILLED AND THE PACKET IS INSERTED ONTO INPTQ.
C  IF BUFCOUNT .GT. MAXCOUNT, THE SAMPLING SWEEP IS FINISHED AND A
C  HALT PACKET IS INSERTED ONTO INPTQ.
C
        INCLUDE         'XFDEF.FOR/NOLIST'
        PARAMETER       MAXCOUNT = 10   !NUMBER OF BUFFERS IN SWEEP
        PARAMETER       ILOGSIZ = 4     !SIZE OF INPUT LOG MESSAGE ARRAY
        PARAMETER       BUFSIZ = 1024   !SIZE OF EACH BUFFER (IN WORDS)
        PARAMETER       NUMBUF = 8      !NUMBER OF BUFFERS
        INTEGER*2       INDEX           !REFERS TO A BUFFER IN BUFARRAY
        INTEGER*2       FUNC            !FUNCTION CODE FROM PACKET
        INTEGER*2       BUFCOUNT        !COUNTS NUMBER OF BUFFERS FILLED
        INTEGER*2       BUFARRAY(BUFSIZ,NUMBUF) !THE ARRAY OF BUFFERS
        INTEGER*4       ACTPARM         !ACTION PARAMETER (NOT USED)
        INTEGER*4       STATUS          !STATUS OF XF$GETPKT (NOT USED)
        INTEGER*4       STAT            !STATUS OF CALL TO XF$PKTBLD
        INTEGER*4       CONTXT(30)      !CONTEXT ARRAY USED BY SUPPORT
        INTEGER*4       ILOGMSG(ILOGSIZ)!STORES LOG MESSAGES FROM DEVICE
        LOGICAL*1       DEVFLAG         !NOT USED IN THIS EXAMPLE
        LOGICAL*1       LOGFLAG         !SIGNALS LOG MESSAGE PRESENT
        COMMON  /MAIN_ACTION/   BUFARRAY,ILOGMSG,BUFCOUNT
        EXTERNAL        SS$_NORMAL
        EXTERNAL        AST$HALT
C
C  PROCESS THE BUFFER
C
        DO 10   I = 1, BUFSIZ
************************************************************************
C
C  AT THIS POINT INSERT THE CODE TO PROCESS ELEMENT (I,INDEX) OF
C  BUFARRAY
C
************************************************************************
10      CONTINUE
```

(Continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
**************************************************************************
C
C  AT THIS POINT INSERT THE CODE TO LOOK AT THE LOG MESSAGE
C
**************************************************************************
C
C  IS THIS THE LAST BUFFER IN THE SWEEP?
C
BUFCOUNT = BUFCOUNT + 1
        IF (BUFCOUNT .LT. MAXCOUNT) THEN        !BUILD A PACKET TO
                                                !REFILL THE BUFFER
        CALL FAKE$PKTBLD (                      !NEED INTERVENING ROUTINE
        1        CONTXT,                        !THE CONTEXT ARRAY
        1       XF$K_PKT_RDCHN,                 !READ DATA CHAINED
        1       INDEX,                          !BUFFER INDEX
        1       ,,,                             !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,                      !SPACE FOR LOG MESSAGE
        1       XF$K_PKT_UNCOND                 !MODES: UNCONDITIONAL
                                                !       INTERRUPT
        1       + XF$K_PKT_CB                   !     : SEND CONTROL BYTE
        1       + XF$K_PKT_INSTL,               !     : INSERT AT TAIL
        1       ,,                              !ACTION GIVEN IN FAKE$PKTBLD
        1       STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))
        ELSE IF (BUFCOUNT .EQ. MAXCOUNT) THEN   !END OF CHAIN
                CALL FAKE$PKTBLD (              !NEED INTERVENING ROUTINE
        1       CONTXT,                         !THE CONTEXT ARRAY
        1       XF$K_PKT_RD,                    !READ DATA FUNCTION
        1       INDEX,                          !BUFFER INDEX
        1       ,,,                             !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,                      !SPACE FOR LOG MESSAGE
        1       XF$K_PKT_UNCOND                 !MODES: UNCONDITIONAL
                                                !       INTERRUPT
        1       + XF$K_PKT_CB                   !     : SEND CONTROL BYTE
        1       + XF$K_PKT_INSTL,               !     : INSET AT TAIL
        1       ,,                              !ACTION GIVEN IN FAKE$PKTBLD
        1       STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))
        ELSE                                    !BUILD A HALT PACKET
                CALL XF$PKTBLD (
        1       CONTXT,                         !THE CONTEXT ARRAY
        1       XF$K_PKT_HALT,                  !ALL DONE
        1       ,,,,                            !DEFAULT VALUES
        1       ILOGSIZ*1,                      !SPACE FOR INPUT LOG MESSAGE
        1       AST$HALT,                       !ACTION ROUTINE
        1       ,                               !NO ACTPARM
        1       STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))
        END IF
        RETURN
        END
```

(Continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
************************************************************************
C
C  PASS ADDRESS OF ACTION ROUTINE TO COMMAND PACKET
C
************************************************************************
        SUBROUTINE      FAKE$PKTBLD(A,B,C,D,E,F,G,H,I,J,K)

C
C  AST$PROCBUF CALLS THIS SUBROUTINE IN ORDER TO PASS THE ADDRESS OF
C  AST$PROCBUF TO XF$PKTBLD.  (AST$PROCBUF CANNOT REFER TO ITSELF
C  WITHIN THE SCOPE OF AST$PROCBUF)
C
        EXTERNAL        AST$PROCBUF
        CALL XF$PKTBLD (A,B,C,D,E,F,G,H,AST$PROCBUF,J,K)
        RETURN
        END
************************************************************************
C
C  HALT ACTION ROUTINE
C
************************************************************************
        SUBROUTINE      AST$HALT (CONTXT,ACTPARM,DEVFLAG,LOGFLAG,
                                  FUNC,INDEX,STATUS)

C
C  THIS IS THE ACTION ROUTINE CALLED BY XF$GETPKT WHEN IT REMOVES A
C  HALT PACKET FROM TERMQ.  THIS ROUTINE PRINTS STATUS INFORMATION,
C  CALLS XF$CLEANUP TO PERFORM FINAL HOUSEKEEPING FUNCTIONS, AND SETS
C  THE EVENT FLAG THAT SIGNALS THE TRANSFER IS COMPLETE.
C
        PARAMETER       EFN = 0
        INTEGER*2       FUNC            !NOT USED
        INTEGER*2       INDEX           !NOT USED
        INTEGER*4       ACTPARM         !NOT USED
        INTEGER*4       STATUS          !NOT USED
        INTEGER*4       STAT            !RETURN FROM XF$CLEANUP
        INTEGER*4       CONTXT(30)      !CONTEXT ARRAY USED BY SUPPORT
        LOGICAL*1       DEVFLAG         !NOT USED
        LOGICAL*1       LOGFLAG         !SIGNALS LOG MESSAGE
        EXTERNAL        SS$_NORMAL      !SUCCESS STATUS RETURN
C
C  PRINT FINAL STATUS
C
        PRINT *,  'FINAL STATUS IN I/O STATUS BLOCK'
        PRINT *,  CONTXT(1), CONTXT(2)

C
C  CLEAN UP
C
        CALL XF$CLEANUP (CONTXT,STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))
        CALL SYS$SETEF (%VAL(EFN))
        RETURN
        END
```

4.7.2 DR32 Queue I/O Functions Program

This sample program (Example 4–2) uses Queue I/O functions to send a device message to the far-end DR-device and then waits for a message returned in a command packet on FREEQ. The returned message is copied into another command packet and that packet writes a data buffer to the far-end DR-device.

Example 4–2 DR32 Queue I/O Functions Program Example

```
; ************************************************************************
;
;                  DR32 QUEUE I/O FUNCTIONS PROGRAM
;
; ************************************************************************
        .TITLE  DR32 PROGRAMMING EXAMPLE
        .IDENT  /01/
;
; DEFINE SYMBOLS
;
        $XFDEF
;
;
; QRETRY - THIS MACRO EXECUTES AN INTERLOCKED QUEUE INSTRUCTION AND
;          RETRIES THE INSTRUCTION UP TO 25 TIMES IF THE QUEUE IS
;          LOCKED.
;
; INPUTS:
;
;       OPCODE = OPCODE NAME: INSQHI,INSQTI,REMQHI,REMQTI
;       OPERAND1 = FIRST OPERAND FOR OPCODE
;       OPERAND2 = SECOND OPERAND FOR OPCODE
;       SUCCESS = LABEL TO BRANCH TO IF OPERATION SUCCEEDS
;       ERROR = LABEL TO BRANCH TO IF OPERATION FAILS
;
; OUTPUTS:
;
;       R0 = DESTROYED
;
;       C-BIT = CLEAR IF OPERATION SUCCEEDED
;               SET IF OPERATION FAILED - QUEUE LOCKED
;               (MUST BE CHECKED BEFORE V-BIT OR Z-BIT)
;
;       REMQTI OR REMQHI:
;
;               V-BIT = CLEAR IF AN ENTRY REMOVED FROM QUEUE; SET
;                       IF NO ENTRY REMOVED FROM QUEUE.
;
;       INSQTI OR INSQHI:
;
;               Z-BIT = CLEAR IF ENTRY IS NOT FIRST IN QUEUE; SET
;                       IF ENTRY IS FIRST IN QUEUE.
;
        .MACRO  QRETRY  OPCODE,OPERAND1,OPERAND2,SUCCESS,ERROR,?LOOP,
                        ?OK
        CLRL    R0
LOOP:
        OPCODE  OPERAND1,OPERAND2
        .IF     NB      SUCCESS
        BCC     SUCCESS
        .IFF
        BCC     OK
        .ENDC
        AOBLSS  #25,R0,LOOP
        .IF     NB      ERROR
        BRW     ERROR
        .ENDC
OK:
        .ENDM   QRETRY
```

(Continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
;
; ALLOCATE STORAGE FOR DATA STRUCTURES
;
        .PSECT  DATA,QUAD
CMDBLK:                                 ; COMMAND BLOCK
INPTQ:  .BLKQ   1                       ; INPUT QUEUE
TERMQ:  .BLKQ   1                       ; TERMINATION QUEUE
FREEQ:  .BLKQ   1                       ; FREE QUEUE
MSGPKT:                                 ; THIS PACKET SENDS A 12-BYTE
                                        ; DEVICE MESSAGE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   12                      ; LENGTH OF DEVICE MESSAGE
        .BYTE   0                       ; LENGTH OF LOG AREA
        .BYTE   XF$K_PKT_WRTCM          ; COMMAND  = WRITE CONTROL
                                        ; MESSAGE
        .BYTE   XF$K_PKT_NOINT@-        ; PACKET CONTROL = NO
                                        ; INTERRUPT
                XF$V_PKT_INTCTL
        .BLKL   1                       ; BYTE COUNT
        .BLKL   1                       ; BUFFER ADDRESS
        .BLKL   2                       ; RESIDUAL MEMORY AND DDI BYTE
                                        ; COUNTS
        .BLKL   1                       ; DR32 STATUS LONGWORD
        .LONG   11111,22222,33333       ; DEVICE MESSAGE
        .LONG   0                       ; EXTEND DEVICE MESSAGE TO
                                        ; QUADWORD LENGTH
        .ALIGN  QUAD
WRTPKT:                                 ; THIS PACKET DOES A WRITE
                                        ; DEVICE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   4                       ; LENGTH OF DEVICE MESSAGE
        .BYTE   0                       ; LENGTH OF LOG AREA
        .BYTE   XF$K_PKT_WRT            ; COMMAND = WRITE
        .BYTE   <XF$K_PKT_CBDMBC@-      ; PACKET CONTROL = SEND
                                        ; COMMAND BYTE,
                XF$V_PKT_CISEL>!-       ; DEVICE MESSAGE, AND BYTE
                                        ; COUNT
                <XF$K_PKT_NOINT@-       ; AND NO INTERRUPT
                XF$V_PKT_INTCTL>
        .LONG   1000                    ; BYTE COUNT
        .LONG   WRTBFR                  ; BUFFER ADDRESS
        .BLKL   2                       ; RESIDUAL MEMORY AND DDI BYTE
                                        ; COUNTS
        .BLKL   1                       ; DR32 STATUS LONGWORD
WDVMSG: .BLKQ   1                       ; SPACE FOR DEVICE MESSAGE
        .ALIGN  QUAD
HLTPKT:                                 ; THIS PACKET HALTS THE DR32
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   0,0,XF$K_PKT_HALT,0     ; COMMAND = HALT
        ,BLKL   5                       ; UNUSED FIELDS IN THIS PACKET
        .ALIGN QUAD
```

(Continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
FREPKT:                                 ; PACKET FOR FREE QUEUE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   4,0,0,0                 ; LENGTH OF DEVICE MESSAGE
                                        ; FIELD
        .BLKL   4                       ; UNUSED FIELDS IN THIS PACKET
        .BLKL   1                       ; DR32 STATUS LONGWORD
        .BLKQ   1                       ; SPACE FOR DEVICE MESSAGE
CMDBLKSIZ=.-CMDBLK
BFRBLK:                                 ; BUFFER BLOCK
WRTBFR: .BLKB   1000
BFRBLKSIZ=.-BFRBLK
CMDTBL: .LONG   CMDBLKSIZ               ; COMMAND BLOCK SIZE
        .LONG   CMDBLK                  ; COMMAND BLOCK ADDRESS
        .LONG   BFRBLKSIZ               ; BUFFER BLOCK SIZE
        .LONG   BFRBLK                  ; BUFFER BLOCK ADDRESS
        .LONG   PKTAST                  ; PACKET AST ADDRESS
        .LONG   0                       ; PACKET AST PARAMETER
        .BYTE   236,XF$M_CMT_SETRTE,0,0 ; DATA RATE (2.0 MBYTES/SEC)
        .LONG   GOBITADR                ; ADDRESS TO STORE THE GO
                                        ; BIT ADDRESS
GOBITADR:
        .BLKL   1
XFIOSB: .BLKL   2                       ; I/O STATUS BLOCK
XFNAMEDSC:
        .LONG   XFNAMESIZ               ; NAME DESCRIPTOR
        .LONG   XFNAME
XFCHAN: .BLKW   1                       ; CHANNEL NUMBER
XFNAME: .ASCII  /XFA0/
XFNAMESIZE=.-XFNAME
; ************************************************************************
;
; PROGRAM STARTING POINT
;
; ************************************************************************
        .PSECT  CODE,NOWRT
        .ENTRY  DREXAMPLE,M<R2,R3>
        $ASSIGN_S DEVNAM = XFNAMEDSC,-  ; ASSIGN A CHANNEL TO DR32
                  CHAN = XFCHAN
        BLBS    R0,10$                  ; SUCCESSFUL ASSIGN
        BRW     ERROR
10$:    MOVAB   CMDBLK,R2
        CLRQ    (R2)+                   ; INITIALIZE INPTQ
        CLRQ    (R2)+                   ; INITIALIZE TERMQ
        CLRQ    (R2)                    ; INITIALIZE FREEQ
;
; INSERT COMMAND PACKET ONTO FREEQ FOR RETURN MESSAGE
;
        QRETRY  ERROR=BADQUEUE,-
        INSQTI  FREPKT,FREEQ
```

(Continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
;
; START DEVICE
;
        $QIO_S  FUNC = #IO$_STARTDATA,-
                CHAN = XFCHAN,-
                IOSB = XFIOSB,-
                EFN = #1,-
                P1 = CMDTBL,-
                P2 = #XF$K_CMT_LENGTH
        BLBC    R0,ERROR
;
; SEND MESSAGE TO far-end DR-DEVICE
;
        QRETRY  ERROR=BADQUEUE,-
        INSQTI  MSGPKT,INPTQ
        MOVL    #1,@GOBITADR            ; SET GO BIT
        $WAITFR_S #1                    ; WAIT UNTIL QIO COMPLETES
;
; CHECK FOR SUCCESSFUL COMPLETION
;
        MOVZWL  XFIOSB,R0
        BEQL    BADQUEUE                ; I/O NOT DONE YET - BAD QUEUE
                                        ; ERROR IN AST ROUTINE
        BLBC    R0,ERROR                ; ERROR
        RET                             ; SUCCESSFUL COMPLETION
BADQUEUE:
        MOVZWL  #SS$_BADQUEUEHDR,R0
;
; AN ERROR HAS OCCURRED.  NORMALLY, THE USER MIGHT PERFORM MORE
; EXTENSIVE ERROR CHECKING AT THIS POINT.  IN PARTICULAR, IF THE ERROR
; IS SS$_CTRLERR, SS$_DEVREQERR, OR SS$_PARITY, THE SECOND LONGWORD
; OF THE I/O STATUS BLOCK CAN PROVIDE ADDITIONAL INFORMATION.  IN THIS
; EXAMPLE, THE PROGRAM EXITS WITH THE ERROR STATUS IN R0.
;
;
; COMMAND PACKET AST ROUTINE
;
PKTAST: .WORD   0
NXTPKT: QRETRY  ERROR=70$,-             ; GET NEXT PACKET FROM QUEUE
        REMQHI  TERMQ,R1
        BVC     10$                     ; PACKET OBTAINED FROM QUEUE
        RET                             ; QUEUE IS EMPTY
10$:    BLBC    XF$L_PKT_DSL(R1),50$    ; RETURN IF PACKET ERROR
        BBC     #XF$V_PKT_FREQPK,-      ; RETURN IF PACKET NOT FROM
                XF$L_PKT_DSL(R1),50$    ; FREEQ
```

(Continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
;
; COMMAND PACKET OBTAINED FROM FREEQ.  COPY DEVICE MESSAGE AND QUEUE
; WRITE PACKET.
;
        MOVL    XF$B_PKT_DEVMSG(R1),WDVMSG
        QRETRY  ERROR=70$,-
        INSQTI  WRTPKT,INPTQ
        QRETRY  ERROR=70$,-
        INSQTI  HLTPKT,INPTQ
        MOVL    #1,@GOBITADR            ; SET GO BIT
50$:    RET
;
; BAD QUEUE ERROR IN AST ROUTINE - WAKE UP MAIN LEVEL.  QIO MAY
; OR MAY NOT HAVE COMPLETED.
;
70$:    $SETEF_S  #1                    ; WAKE UP MAIN LEVEL
        RET
        .END    DREXAMPLE
```

5 DUP11 Interface Driver

This section describes the use of the DUP11 device interface driver (XWDRIVER). The DUP11 is the lowest-level user interface to the VAX 2780/3780 protocol emulator. (The user can also access the 2780/3780 through the command language interface and the record-oriented interface. See the *VAX 2780/3780 Protocol Emulator User's Guide.*)

5.1 Supported Device

The DUP11 is a single line, program-controlled communications device that interfaces a VAX processor to a serial, synchronous communications line. Data transmission occurs at a maximum speed of 9600 baud. Although the DUP11 functions in either full- or half-duplex mode, the DUP11 driver operates logically only in half-duplex mode; only one I/O request is processed at any given time, but many may be queued.

The DUP11 driver transfers output data from the VAX/VMS system to the DUP11. The DUP11 then shifts the data onto the communications line. Input data from the communications line modem is shifted into the DUP11, where it is made available to the DUP11 driver on an interrupt basis.

5.1.1 Driver Operating Modes

The device driver functions in two operating modes: binary synchronous communications (BSC) mode and binary mode. BSC mode operations are described in Appendix C of the *VAX 2780/3780 Protocol Emulator User's Guide.* The preface of the same manual also provides a list of related documents.

In BSC mode, the driver observes standard point-to-point BSC protocol in send and receive operations. In binary mode, the driver does not observe any protocol; the only operation performed on the data is the insertion or deletion of padding (PAD) and synchronization (SYN) characters. An operation is completed when the buffer count reaches zero or the I/O is canceled.

Function modifiers, which are included in all read and write requests to the driver, define the operating mode for each I/O operation.

If the only reason for not using the record-oriented interface is the restriction on block size (the application is compatible with all other 2780/3780 communications protocols), the DUP11 driver should be used primarily in BSC mode rather than binary mode. Binary mode is used only if the user requires direct control of some aspect of the communications protocol handled by the driver when in BSC mode. All line protocol messages, for example, bids and end-of-transmission (EOT) signals, must be transmitted in binary mode.

5.1.1.1 BSC Mode

If the IO$K_PTPBSC function modifier is included in a read or write request, data is read or written in BSC mode. The DUP11 driver performs the following operations:

- Inserts in the output data, and removes from the input data, BSC data-link control characters, for example, start of text (STX) and end of intermediate transmission block (ITB).
- Checks input message blocks for transmission errors. Adds cyclic redundancy check (CRC) characters to output message blocks to support error checking by the communications processor in the remote system.
- Manages line protocols, for example, acknowledgment (ACK), negative acknowledgment (NAK), and enquiry (ENQ) responses, that determine whether a message block must be retransmitted because of transmission errors.
- Inserts in the output data, and removes from the input data, data-link escape (DLE) information in transparent mode.

The DUP11 driver does not modify the input or output data in any way. All necessary processing, for example, data translation and space compression or expansion, must be included in the user program. The user program builds the message block to be transmitted into a single buffer. This buffer must start with a two-byte count that includes all data up to the point where a CRC will be placed, and end with a two-byte count field equal to -1. The driver inserts an ITB character in front of internal CRC characters.

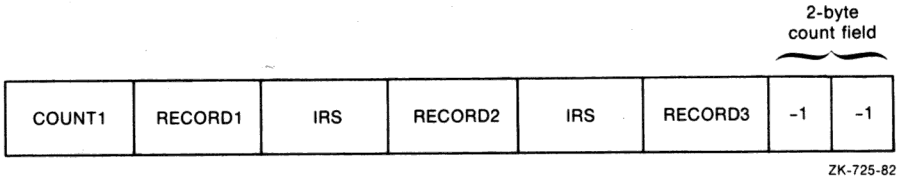
Figures 5–1 through 5–5 illustrate how the DUP11 driver reformats user-formatted output message blocks into standard 2780/3780 message blocks. The driver unblocks input messages in the reverse order of that shown in these figures.

All COUNT and CRC fields in these examples are two bytes long. Each record count results in the generation of a CRC character. An ITB character precedes all internal CRC characters. An ETB precedes the last CRC in a block unless the IO$M_LASTBLOCK function modifier is specified. In that case, an ETX precedes the CRC. If in transparency mode (specified by IO$_SETMODE), all data-link control characters are preceded by a DLE character and all DLE characters in the data buffers are changed to DLE DLE. Also, the control character sequence of SYN, SYN, DLE, STX is inserted between records within the message block.

Message blocks transmitted by the DUP11 driver include a prefix of SYN characters (as specified by the set mode function) and a suffix of a PAD character (hexadecimal FF).

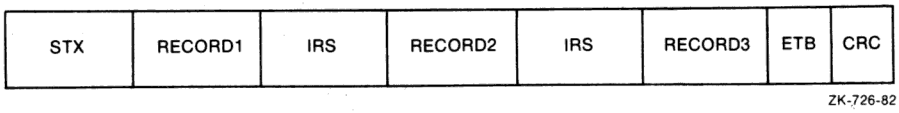
Figure 5–1 shows the format of user-built message buffers that simulate 3780 processing. The user must pass the buffers to the device driver by issuing function requests that include the IO$K_PTPBSC function modifier.

Figure 5–1 3780 Message Block Example



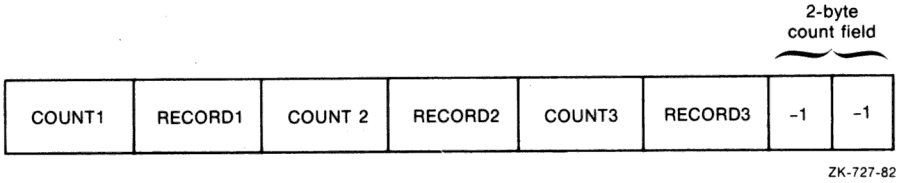
The DUP11 driver transmits the message block after modifying the format, as shown in Figure 5–2. The driver does not modify the data records in the two buffers; they are identical.

Figure 5–2 3780 Message Block Example (Modified)



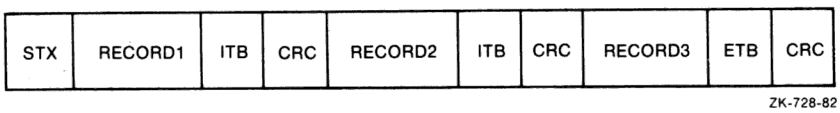
To simulate 2780 processing in nontransparent mode, the user builds message buffers in the format shown in Figure 5–3. The user must include the IO$K_PTPBSC function modifier in the QIO requests that pass the buffers to the DUP11 driver.

Figure 5–3 Nontransparent 2780 Message Block Example



The DUP11 driver transmits the message block after modifying the format, as shown in Figure 5–4.

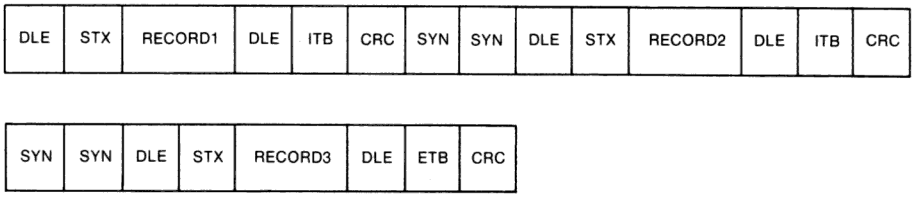
Figure 5–4 Nontransparent 2780 Message Block Example (Modified)



To simulate 2780 processing in transparent mode, the user must specify the transparency modifier in a set mode function request, build message buffers in the format shown in Figure 5–3, and include the IO$K_PTPBSC function modifier in the write function requests that pass the buffers to the DUP11

driver. The driver transmits the message block after modifying the format, as shown in Figure 5–5. The driver adds a duplicate DLE character to any DLE character encountered in the data records.

Figure 5–5 Transparent 2780 Message Block Example (Modified)



5.1.1.2 Binary Mode

If the IO$K_SRRUNOUT function modifier is included in a read or write request, data is read or written in binary mode. In binary mode, the DUP11 driver performs no processing operations on the user-supplied message block buffer. Except for the insertion in output message blocks, and deletion from input message blocks, of leading SYN and trailing PAD characters, data passes through the DUP11 driver as unprocessed, binary information. The user program directly controls all data transmitted or received by the driver. QIO requests in the user program provide all necessary communications to the remote system. The user program must perform the following functions:

- Explicitly issue all protocol messages, for example, ACK, NAK, and ENQ responses, to the DUP11 driver.
- Perform all validity checking calculations and comparisons.
- Handle the insertion and removal of any message-framing and interrecord control characters in the message blocks.
- Repeat write function requests until the operation is successful or the program's error threshold is reached.

5.2 Device Information

Users can obtain information on DUP11 characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume*.)

$GETDVI returns DUP11 characteristics when you specify the item code DVI$_DEVCHAR. Table 5–1 lists these characteristics, which are defined by the $DEVDEF macro.

DVI$_DEVBUFSIZ returns the device buffer size (default is 520 bytes). DVI$_DEVDEPEND returns the line characteristics, the SYN character, and the time in a longword field. (Bytes 0 and 1 = characteristics, byte 2 = SYN, and byte 3 = time.)

The time is the time (in seconds) to wait for a clear to send (CTS) signal. The SYN character is that character selected to precede all message blocks transmitted by the DUP11 driver. Table 5–2 lists the line characteristics. The information returned in DVI$_DEVBUFSIZ and DVI$_DEVDEPEND is established by IO$_SETMODE (see Section 5.3.3).

Table 5–1 Device-Independent Characteristics

Characteristic	Meaning
Dynamic Bits[1] (Conditionally Set)	
XW$M_CHA_FDX	Full-duplex line
XW$M_CHA_XPR	Transparency mode
XW$M_CHA_DSC	Data set ready
Static Bits[2] (Always Set)	
DEV$M_AVL	Available device
DEV$M_IDV	Input device
DEV$M_ODV	Output device

[1]Defined by the $XWDEF macro.

[2]Defined by the $DEVDEF macro.

Table 5–2 DUP11 Line Characteristics

Characteristic	Meaning
XW$M_CHA_DSC	Sense state of data terminal ready (DTR) signal line; meaningful only to IO$_SENSEMODE
XW$M_CHA_FDX	Full-duplex mode; does not drop request to send (RTS) signal after each segment is transmitted
XW$M_CHA_XPR	Transparent mode; used only when IO$K_PTPBSC is specified with a write function

5.3 DUP11 Function Codes

The DUP11 can perform logical and physical I/O operations. The basic I/O functions are read, write, set mode, and sense mode. Table 5–3 lists these functions and their function codes. The following sections describe these functions in greater detail.

Table 5–3 DUP11 I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_READLBLK P1,P2	L	IO$K_SRRUNOUT IO$K_PTPBSC	Read logical block.
IO$_READPBLK P1,P2	P	IO$K_SRRUNOUT IO$K_PTPBSC	Read physical block.

[1]L = logical, P = physical

Table 5–3 (Cont.) DUP11 I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_WRITELBLK P1,P2	L	IO$K_SRRUNOUT IO$K_PTPBSC IO$M_LASTBLOCK[2]	Write logical block.
IO$_WRITEPBLK P1,P2	P	IO$K_SRRUNOUT IO$K_PTPBSC IO$M_LASTBLOCK[2]	Write physical block.
IO$_SETMODE P1	L	IO$M_STARTUP IO$M_NODSRWAIT[3] IO$M_SHUTDOWN	Set line state or line parameters.
IO$_SENSEMODE	L		Sense line state; return status.

[1]L = logical, P = physical

[2]Use only with IO$K_PTPBSC

[3]Use only with IO$M_STARTUP

5.4 Read

Read functions provide for the transfer of data from the DUP11 into the user process's virtual memory address space. The VAX/VMS operating system provides two function codes:

- IO$_READLBLK—read logical block
- IO$_READPBLK—read physical block

The read function codes take two device/function-dependent arguments:

- P1—the starting virtual address of the buffer that is to receive data
- P2—the size of the data buffer in bytes

The read functions can take two function modifiers:

- IO$K_SRRUNOUT—read data in binary format until count runout (see Section 5.1.1.2)
- IO$K_PTPBSC—read data in BSC mode (see Section 5.1.1.1)

5.4.1 Write

Write functions provide for the transfer of data to the DUP11 from the user process's virtual memory address space. The VAX/VMS operating system provides two function codes:

- IO$_WRITELBLK—write logical block
- IO$_WRITEPBLK—write physical block

The write function codes take two device/function-dependent arguments:

- P1—the starting virtual address of the buffer that is to send data to the DUP11
- P2—the size of the data buffer in bytes

The write functions can take three function modifiers:

- IO$K_SRRUNOUT—write data in binary format until count runout (see Section 5.1.1.2)
- IO$K_PTPBSC—write data in BSC mode (see Section 5.1.1.1)
- IO$M_LASTBLOCK—terminate the data block with an ETX character (This function modifier can be used only in conjunction with IO$K_PTPBSC.)

5.4.2 Set Mode

The set mode function is used to change the state of the communication line or the parameters that control the line. The VAX/VMS operating system provides one function code:

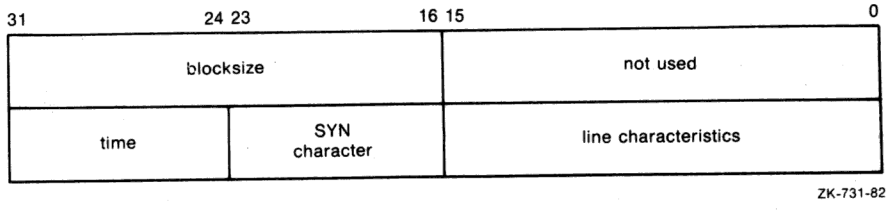
- IO$_SETMODE—set mode

This function code takes the following device/function-dependent argument:

- P1—points to a quadword buffer block that contains the new communication line parameters

Figure 5–6 shows the format of the P1 buffer.

Figure 5–6 Set Mode P1 Buffer



In the first longword, blocksize is the largest buffer expected. This parameter is included in the buffer block only when an IO$_READLBLK request includes the IO$K_PTPBSC function modifier.

The first word of the second longword specifies the following line characteristics:

- XW$M_CHA_DSC—sense state of data terminal ready (DTR) signal line; meaningful only to IO$_SENSEMODE
- XW$M_CHA_FDX—full-duplex mode; does not drop request to send (RTS) after each segment is transmitted
- XW$M_CHA_XPR—transparent mode; used only when IO$K_PTPBSC is specified with a write function

The third byte of the second longword is the SYN character that precedes all message blocks transmitted by the DUP11 driver. The fourth byte specifies the time, in seconds, to wait for a clear to send (CTS) signal. This parameter is included in the buffer block only when a read or write request specifies the IO$K_SRRUNOUT function modifier.

The set mode function can take three function modifiers:

- IO$M_STARTUP—enable the communication line, that is, assert data terminal ready (DTR) and wait for data set ready (DSR) signals
- IO$M_NODSRWAIT—complete this function without regard to the state of DSR (To achieve this result, IO$M_NODSRWAIT must be included in each set mode function.); used only in conjunction with the IO$M_STARTUP function modifier
- IO$M_SHUTDOWN—disable the communication line (disable DTR signal)

5.4.3 Sense Mode

The sense mode function senses the current state of the communication line and returns the line characteristics and status in the I/O status block (see Figure 5–8). The VAX/VMS operating system provides one function code:

- IO$_SENSEMODE

The sense mode function takes no function modifiers.

5.5 I/O Status Block

Figure 5–7 shows the I/O status block for all DUP11 functions except sense mode. Figure 5–8 shows the I/O status block for the sense mode function. Appendix A lists the status returns for all functions. (The *VAX/VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.)

Figure 5–7 IOSB Contents

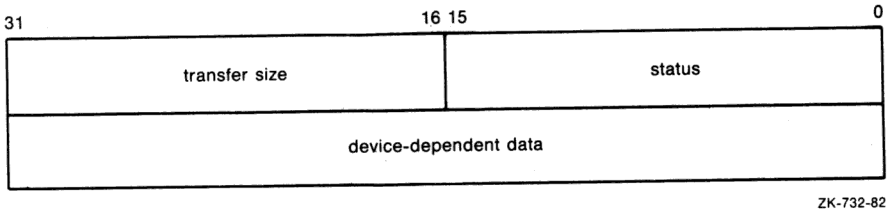
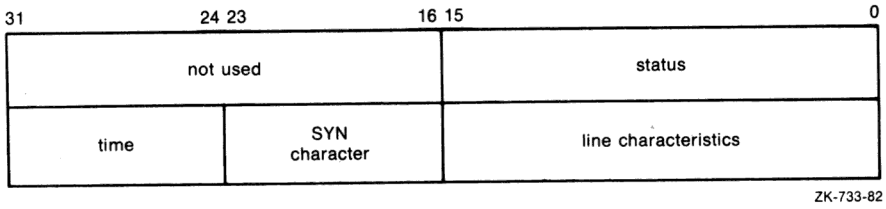


Figure 5–8 IOSB Contents—Sense Mode



In Figure 5–7, the second word of the first longword contains the size of the transfer in bytes. For transmit (write) operations, the transfer size is the value specified in the P2 argument. For read (receive) operations, transfer size is the amount of data received as the result of the read request. Table 5–4 lists the device-dependent data returned in the second longword.

Table 5–4 Device-Dependent Status Returns

Value	Meaning
XW$M_BADCHAIN	A record list was incorrectly found in a BSC (IO$K_PTPBSC) write request. This is a fatal error condition.
XW$M_CONACK	A BSC (IO$K_PTPBSC) write request was completed with a conversational ACK character. The data block is considered acknowledged. However, the data received with the ACK character is lost.
XW$M_DATACK	Retry threshold was exceeded. This is a fatal error condition.
XW$M_DISCON	BSC disconnect sequence was received, that is, DLE, EOT. This is a fatal error condition.
XW$M_EOT	EOT was received. This is a fatal error condition.
XW$M_EXTEND	A BSC (IO$K_PTPBSC) read request was completed successfully. The read data included a block that ended with an EXT character.
XW$M_ILLMOD	Illegal function modifier was detected. This is a fatal error condition.
XW$M_NODSR	Request was aborted because of DSR loss. This is a fatal error condition.

Table 5–4 (Cont.) Device-Dependent Status Returns

Value	Meaning
XW$M_PIPE_MARK	A BSC (IO$K_PTPBSC) transfer was aborted because of a previous failure. This is a fatal error condition.
XW$M_RVI	A BSC (IO$K_PTPBSC) write request was completed with a received RVI.
XW$M_TRABINTMO	A timeout occurred during a binary (IO$K_SRRUNOUT) data transfer. This is a fatal error condition.
XW$M_XPR	A BSC (IO$K_PTPBSC) read request was satisfied with a transparent block. The received information was transmitted (written) in transparency mode.

In Figure 5–8, the first longword contains the current status of the communication line. Appendix A lists the status return values and their meaning.

The first word of the second longword returns one or more of the following line characteristics:

- XW$M_CHA_DSC—state of DTR line
- XW$M_CHA_FDX—full-duplex mode; does not drop RTS after each segment transmitted
- XW$M_CHA_XPR—transparent mode; used only when IO$K_PTPBSC is specified with a write function

The third byte of the second longword is the SYN character selected to precede all message blocks transmitted by the DUP11 driver. The fourth byte specifies the time, in seconds, to wait for the clear to send (CTS) signal. This parameter is included in the buffer block only when the IO$K_SRRUNOUT function modifier is specified in a read or write request.

6 DEUNA, DEQNA, and DELUA Device Drivers

This section describes the QIO interface of the DEUNA (DIGITAL Ethernet UNIBUS Adapter), DEQNA (DIGITAL Ethernet Q-bus Adapter), and DELUA (DIGITAL Ethernet LSI to UNIBUS Adapter) interface communications drivers. The DEUNA and DELUA use the XEDRIVER; the DEQNA uses the XQDRIVER.

Note: Unless otherwise stated, references to the DEUNA also apply to the DEQNA and DELUA.

All devices support Ethernet and Institution for Electrical and Electronic Engineers (IEEE) 802 standards, except where otherwise indicated. Section 6.1.4 lists the specific IEEE 802 features supported by the driver.

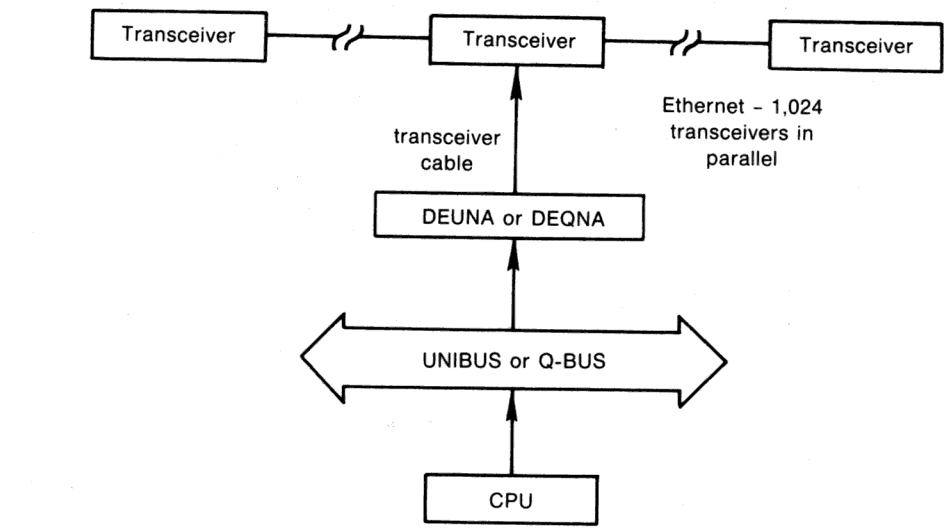
6.1 Supported Devices

The DEUNA is a direct-memory-access (DMA) device that, with the H4000 transceiver, implement the Ethernet specification. A single DEUNA is a controller, which is a piece of peripheral equipment of the system bus that communicates with the local system and with remote systems implementing the Ethernet specification. The Ethernet specification is described in *The Ethernet–Data Link Layer and Physical Layer Specification (No. AA-K759B-TK).*

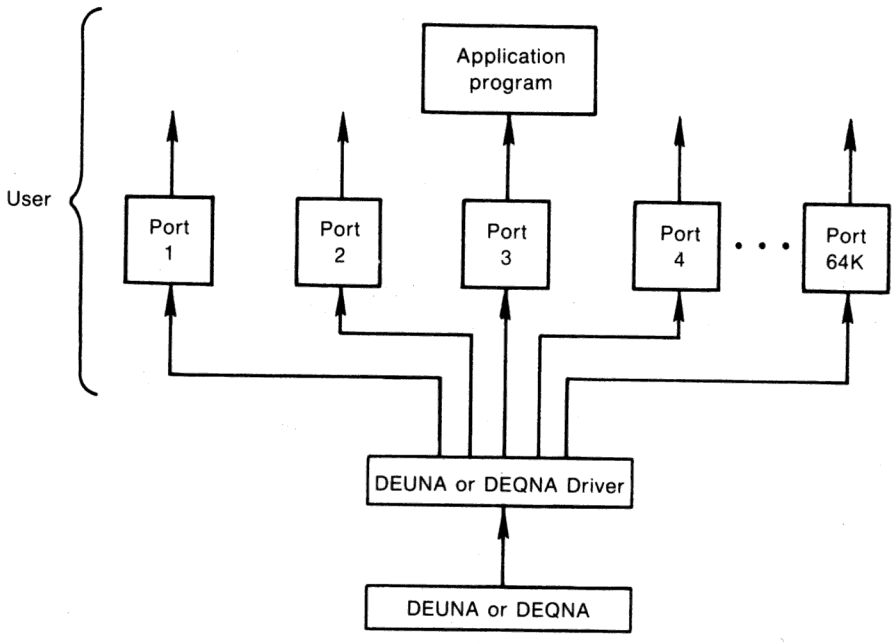
The DEUNA uses a single multiaccess channel with carrier sense and collision detection (CSMA/CD) to provide direct communication between a VAX processor and the Ethernet. The Ethernet is that group of DIGITAL products that implement XEROX, INTEL, and DIGITAL intercompany Ethernet specifications. A *port* in a DEUNA configuration consists of a protocol type and a controller (DEUNA). A *protocol type* is a unique 16-bit address that identifies each user of the DEUNA (see Sections 6.1.2 and 6.1.3). There are as many ports on a DEUNA as there are protocol types. Each protocol type is independent of other protocol types running on the same DEUNA.

Application programs use the DEUNA driver's QIO interface to perform I/O operations to and from another device on the Ethernet. This chapter describes the QIO interface. Figure 6–1 shows the relationship of the DEUNA to the processor and the user application program.

Figure 6–1 Typical DEUNA Configuration



a. Hardware Interface



b. Software Interface

ZK-1129-82

6.1.1 Driver Initialization and Operation

DIGITAL recommends that users perform the following sequence to initialize and start the DEUNA device driver:

1. Assign an I/O channel to XEc (DEUNA and DELUA) or XQc (DEQNA) with the Assign I/O Channel ($ASSIGN) system service, where c is the DEUNA controller through which the data transfer will occur. $ASSIGN creates a new unit control block (UCB) to which the channel for the DEUNA port is assigned. The user can now define the mode and, if desired, assign other channels to this UCB.
2. Start up the DEUNA port with a set mode function (see Section 6.3.3.1). The user must supply the required P2 buffer parameters.
3. Perform read, write, and sense mode operations as desired.
4. Shut down the DEUNA port with a set mode function (see Section 6.3.3.3).
5. Deassign the I/O channel with the Deassign I/O Channel ($DASSGN) system service.

6.1.2 Ethernet Addresses

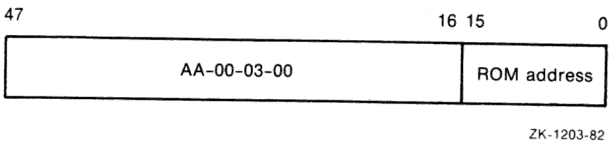
The Ethernet is a medium for creating a network; it is not a network by itself. The DEUNA device and the local system constitute a node. Nodes on Ethernet lines are identified by unique Ethernet addresses. A message can be sent to one, several, or all nodes on an Ethernet line simultaneously, depending on the Ethernet address used. You do not have to specify the Ethernet address of your own node to communicate with other addresses on the same node. However, you do need to know the Ethernet address of the node with which you wish to communicate.

6.1.2.1 Format of Ethernet Addresses

An Ethernet address is 48 bits in length. Ethernet addresses are represented by the Ethernet standard as six pairs of hexadecimal digits (six bytes), separated by hyphens (for example, AA-01-23-45-67-FF). The bytes are displayed from left to right in the order in which they are transmitted; bits within each byte are transmitted from right to left. In the example, byte AA is transmitted first; byte FF is transmitted last. (See the description of NMA$C_PCLI_PHA in Table 6–5 for the internal representation of addresses.)

Xerox Corporation assigns a block of addresses to a producer of Ethernet interfaces upon application. Thus every manufacturer has a unique set of addresses to use. Normally, one address out of the assigned block of physical addresses is permanently associated with each interface (usually in read-only memory). This address is known as the Ethernet hardware address of the interface.

DIGITAL's interface to Ethernet (the DEUNA controller at the node) can set a different logical address to be used by the interface: this address is known as the Ethernet physical address. On powerup of the node, the physical address is set to the hardware address. Specifically, the DEUNA constructs the Ethernet physical address by appending a 16-bit read-only memory (ROM) address to a constant 32-bit number (AA-00-03-00) within the block of Ethernet addresses assigned to DIGITAL:



An example is a DEUNA with ROM address 182 (decimal), which would be set to an Ethernet physical address of AA-00-03-00-B6-00. Because the DEUNA at the node constructs its own physical address, users normally do not need to manipulate Ethernet addresses directly.

Once the Ethernet physical address has been set to its new value, it is reset to its original hardware address value only under the following circumstances:

- When a reset is issued to the DEUNA (for example, when the machine power is shut off)
- When the state of the Ethernet line is set to OFF

6.1.2.2 Ethernet Multicast Addresses

An Ethernet address can be a physical address of a single node or a multicast address, depending on the value of the low-order bit of the first byte of the address (this bit is transmitted first). The two types of node addresses are:

- Physical address—the unique address of a single node on any Ethernet (as described previously). The least significant bit of the first byte of an Ethernet physical address is 0. (For example, in physical address AA-00-03-00-FC-00, byte AA in binary is 1010 1010, and the value of the low-order bit is 0.)
- Multicast address—a multidestination address of one or more nodes on a given Ethernet. The least significant bit of the first byte of a multicast address is 1. (For example, in the multicast address AB-22-22-22-22-22, byte AB in binary is 1010 1011, and the value of the low-order bit is 1.) Multicast addresses can be either of the following:
  - Multicast group address—any number of node groups can be assigned a group address so that they are all able to receive the same message in a single transmission by a sending node.
  - Broadcast address—a single multicast address (specifically, FF-FF-FF-FF-FF-FF) that can be received by all nodes on a given Ethernet. (Note that the broadcast address should be used only for messages to be acted on by all nodes on the Ethernet, since all nodes must process them.)

6.1.2.3 DIGITAL Ethernet Physical and Multicast Address Values

DIGITAL physical addresses are in the range AA-00-00-00-00-00 through AA-00-04-FF-FF-FF. DIGITAL multicast addresses assigned for use in cross-company communications are:

Value	Meaning
FF-FF-FF-FF-FF-FF	Broadcast
CF-00-00-00-00-00	Loopback assistance

DIGITAL multicast addresses assigned to be received by other DIGITAL nodes on the same Ethernet are:

Value	Meaning
AB-00-00-01-00-00	Dump/load assistance
AB-00-00-02-00-00	Remote console
AB-00-00-03-00-00	All Phase IV routers
AB-00-00-04-00-00	All Phase IV end nodes
AB-00-00-05-00-00 through AB-00-03-FF-FF-FF	Reserved for future use
AB-00-04-00-00-00 through AB-00-04-FF-FF-FF	For use by DIGITAL customers for their own applications

DECnet–VAX always sets up the DEUNA at each node to receive messages sent to any address in the preceding list of DIGITAL multicast addresses.

6.1.3 Ethernet Protocol Types

Every Ethernet frame has a 16-bit protocol field. This field is used to allow multiple users of Ethernet at a single station. Protocol types are independent of addresses; Xerox Corporation is also responsible for assigning unique protocol designations to interested parties. Whenever an Ethernet user at a particular station turns on a circuit, that user must specify the protocol type to be used on that circuit; messages sent over that circuit always have the protocol type attached by the DEUNA device driver, and messages received with that protocol type are delivered to the starter of that circuit. Note that, since multicast addresses are enabled on a line basis and protocol types are enabled on a circuit basis, a station may receive a message that no user can process. DIGITAL's protocol types are in the range 60-00 through 60-09.

The cross-company protocol type is:

Value	Meaning
90-00	Loopback assistance

DIGITAL's protocol types are:

Value	Meaning
60-01	Dump/load assistance
60-02	Remote console
60-03	Routing
60-04	LAT
60-06	For use by DIGITAL customers for their own applications

6.1.4 IEEE 802 Support

The DEUNA driver supports the following IEEE 802 features:

- IEEE 802.2 packet format and IEEE 802.3 packet format
- IEEE 802.2 Class I service including the UI, XID, and TEST commands and the XID and TEST responses (Class II service must be provided by the user.)
- Six-byte destination and source address fields

  The IEEE 802.3 Standard states that the size of the destination and source addresses may be two or six bytes, as decided by the manufacturer. The DEUNA driver does not support two-byte address fields.
- Physical layer identified as type 10BASE5 (10 megabytes/second baseband medium with maximum segment length of 500 megabytes)

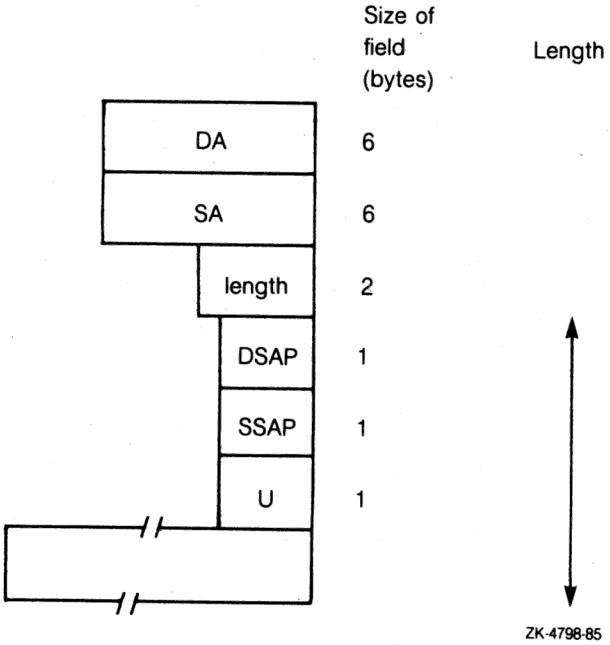
6.1.5 IEEE 802 Packet Format

The IEEE 802 packet formats accepted for a channel depend on the service on that channel.

6.1.5.1 Class I Service Packet Format

For Class I service, only three packet formats are transmitted and received: UI, XID, and TEST. Figure 6–2 shows the format of these packets.

Figure 6–2 Class I Service Packet Format



The field definitions for the Class I service packet are as follows:

- DA—destination address
- SA—source address
- LENGTH—length of the 802.3 frame

- DSAP—destination service access point (SAP)
- SSAP—source SAP
- U—unnumbered control field command/response
- DATA—user-supplied data plus padding (PAD)

The unnumbered control field (U), which is always one byte in length, can be one of the following binary values:

- UI command (00000011)

  This is the unnumbered information command. It is the method used to transmit data from one user to another and is the most widely used control field value.

  The UI command can be specified by using NMA$C_CTLVL_UI.

- XID command (101p1111)

  This is the exchange identification command. It is used to convey information. The "p" bit is the poll bit and may be either 0 or 1. This command can be specified by using NMA$C_CTLVL_XID for a "0" poll bit or NMA$C_CTLVL_XID_P for a "1" poll bit.

- XID response (101f1111)

  The XID response is a response to an XID command. The "f" bit is the final bit and will match the poll bit from the XID command.

- TEST command (111p0011)

  The TEST command is used to test a connection. The "p" bit is the poll bit and may be either 0 or 1. This command can be specified by using NMA$C_CTLVL_TEST for a "0" poll bit or NMA$C_CTLVL_TEST_P for a "1" poll bit.

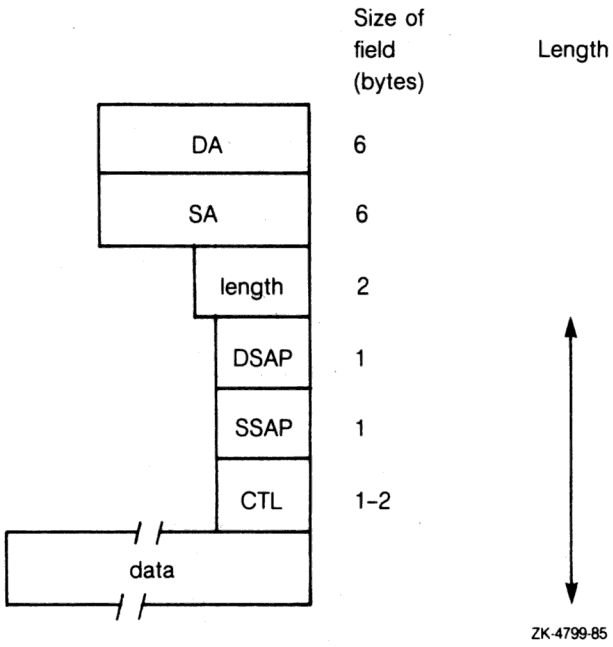
- TEST response (111f0011)

  The TEST response is a response to a TEST command. The "f" bit is the final bit and will match the poll bit from the TEST command.

6.1.5.2 User-Supplied Service Packet Format

Figure 6–3 shows the packet format for user-supplied service.

Figure 6–3 User-Supplied Service Packet Format



The field definitions for the user-supplied service packet are as follows:

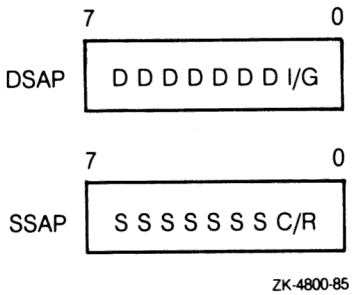
- DA—destination address
- SA—source address
- LENGTH—length of the 802.3 frame
- DSAP—destination SAP
- SSAP—source SAP
- CTL—control field
- DATA—user-supplied data plus PAD

The user provides the control field values, which are documented in the IEEE 802.2 Standard. The user-supplied packet format is the generic packet format as specified in the IEEE 802.2 Standard. Class I packets (see Section 6.1.5.1) are simply a specific version of this generic packet format. Therefore, if the control field value of the user-supplied packet is UI, XID, or TEST, then the packet is the same as a Class I packet. The user should note, as defined in the IEEE 802.2 Standard, that Class II packets include the UI, XID, and TEST command/response formats.

6.1.6 Service Access Point (SAP) Restrictions

The IEEE 802.2 Standard places restrictions on both user SAPs and SAPs that can be used as source SAPs (SSAP). All SAPs are eight bits long. Figure 6–4 shows the format of DSAPs and SSAPs.

Figure 6–4 DSAP and SSAP Format



Use of the least significant bit depends on whether the SAP is a source SAP (SSAP) or a DSAP. For a DSAP field, the least significant bit distinguishes group SAPs (bit 0 = 1) from individual SAPs (bit 0 = 0). For an SSAP field, the least significant bit distinguishes commands (bit 0 = 0) from responses (bit 0 = 1). Because these two bits are located at the same bit position within the SAP field, a group SAP cannot be used as an SSAP. If this were allowed, a group SAP would be interpreted as an individual SAP with the command/response bit set to 1, thus implying a response.

The IEEE 802.2 Standard reserves for its own definition all SAP addresses with the second least significant bit set to 1.

6.2 Device Information

Users can obtain information on DEUNA characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VAX/VMS System Services Reference Manual* in the *VAX/VMS System Routines Reference Volume*.)

$GETDVI returns DEUNA characteristics when you specify the item code DVI$_DEVCHAR. Table 6–1 lists these characteristics, which are defined by the $DEVDEF macro.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and device class names, which are defined by the $DCDEF macro. The device type is DT$_DEUNA for the DEUNA, DT$_DEQNA for the DEQNA, and DT$_DELUA for the DELUA. The device class for the DEUNA is DC$_SCOM. DVI$_DEVBUFSIZ returns the maximum message size. The maximum send or receive message size is 1500 bytes if padding (NMA$C_PCLI_PAD) is not enabled, or 1498 bytes if padding is enabled.

Table 6–1 DEUNA, DEQNA, and DELUA Device Characteristics

Characteristic[1]	Meaning
Static Bits (Always Set)	
DEV$M_NET	Network device
Static Bits (Always Set)	**Meaning**
DEV$M_ODV	Output device
DEV$M_IDV	Input device
DEV$M_SHR	Shareable device

[1]Defined by the $DEVDEF macro.

DVI$_DEVDEPEND returns the unit status and an error summary in a longword field. (Byte 1 = status and byte 2 = error summary; bytes 0 and 3 are not used.) The status bits show the status of the unit and the line. They can be set or cleared only when the controller is not active.

Table 6–2 lists the status values and their meanings. These values are defined by the $XMDEF macro.

Table 6–2 DEUNA Unit and Line Status

Status	Meaning
XM$M_STS_ACTIVE	Protocol is active.
XM$M_STS_BUFFAIL	Attempt to allocate a receive buffer failed.
XM$M_STS_TIMO	Timeout occurred on DEUNA.

The error summary bits are set when an error occurs. They are read-only bits. If an error is fatal, the Ethernet port is shut down. Table 6–3 lists the error summary bit values and their meanings.

Table 6–3 Error Summary Bits

Error Summary Bit	Meaning
XM$M_ERR_FATAL	Hardware or software error occurred on DEUNA port.
XM$M_ERR_LOST	Data was lost when the message received was longer than the specified maximum message size.

6.3 DEUNA Function Codes

The DEUNA driver can perform logical, virtual, and physical I/O operations. The basic functions are read, write, set mode, set characteristics, and sense mode. Table 6–4 lists these functions and their codes. The following sections describe these functions in greater detail.

Table 6–4 DEUNA I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_READLBLK P1,P2,-[P5]	L	IO$M_NOW	Read logical block.
IO$_READVBLK P1,P2,-[P5]	V	IO$M_NOW	Read virtual block.
IO$_READPBLK P1,P2,-[P5]	P	IO$M_NOW	Read physical block.
IO$_WRITELBLK P1,P2,-[P4],P5	L		Write logical block.
IO$_WRITEVBLK P1,P2,-[P4],P5	V		Write virtual block.
IO$_WRITEPBLK P1,P2,-[P4],P5	P		Write physical block.
IO$_SETMODE P1,[P2],-P3[2]	L	IO$M_CTRL IO$M_STARTUP IO$M_SHUTDOWN IO$M_ATTNAST	Set DEUNA characteristics and controller state for subsequent operations.
IO$_SETCHAR P1,[P2],-P3[2]	P	IO$M_CTRL IO$M_STARTUP IO$M_SHUTDOWN IO$M_ATTNAST	Set DEUNA characteristics and controller state for subsequent operations.
IO$_SENSEMODE [P1],-[P2]	L	IO$M_CTRL	Sense controller characteristics and return them in specified buffer(s).

[1]V = virtual, L = logical, P = physical (There is no functional difference in these operations.)

[2]The P1 and P3 arguments are only for attention AST QIOs.

Although the DEUNA device driver does not differentiate among logical, virtual, and physical I/O functions (all are treated identically), the user must have the required privilege to issue the request.

6.3.1 Read

Read functions provide for the direct transfer of data from another port on the Ethernet into the user process's virtual memory address space. The VAX/VMS operating system provides three function codes:

- IO$_READLBLK—read logical block
- IO$_READVBLK—read virtual block
- IO$_READPBLK—read physical block

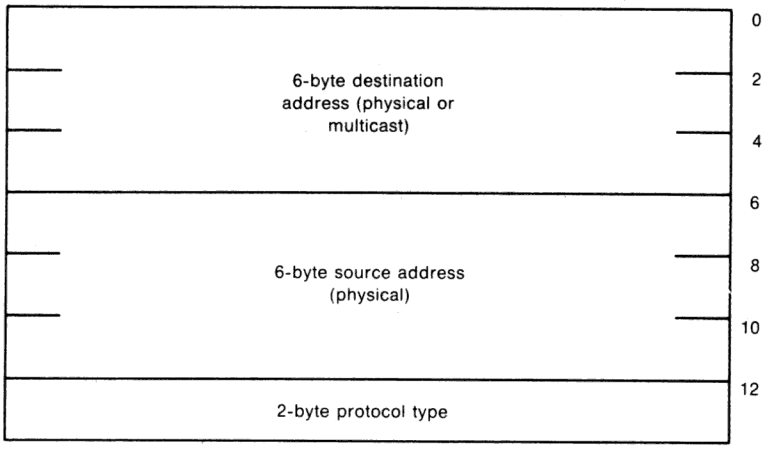
Received messages are multibuffered in system-dynamic memory and then copied to the user's buffer when a read operation is performed.

The read functions take three device/function-dependent arguments:

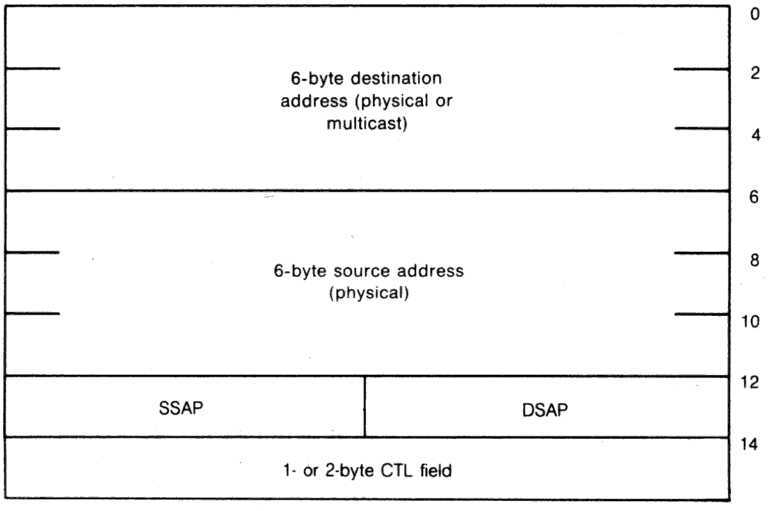
- P1—the starting virtual address of the buffer that is to receive data
- P2—the size of the receive buffer in bytes
- P5—the address of either a 14-byte buffer that will contain the destination address (either physical or multicast), the source address (physical), and the protocol type for an Ethernet format packet, or a 16-byte buffer that will contain the destination address, the source address, the DSAP, the SSAP, and the CTL field value for an 802 format packet.

  The format of the buffer is:

Ethernet Format:



IEEE 802 Format:



ZK-1126-82

The message size specified by P2 cannot be larger than 1500 bytes (see Section 6.2). If a message larger than the maximum size is received, a status of SS$_DATAOVERUN is returned in the I/O status block. The P1 and P2 arguments must always be specified; the P5 argument is optional. However,

if P5 is not specified, the user will be unable to determine the source of the received message.

For 802 format packets the P5 buffer always contains the DSAP and SSAP in bytes 13 and 14. The next one or two bytes (bytes 15 and 16) following the SSAP contain the control field value. For Class I service, the control field value is always one byte in length and will always be placed in byte 15 of this buffer. For-user supplied service, the user will have to determine the length of the control field value according to the IEEE 802 Standard (see Section 6.1.5.2).

On 802 format packets, the maximum message length depends on the size of the CTL field: for a one-byte CTL field, the maximum message length is 1497 bytes; for a two-byte CTL field, the maximum message length is 1496 bytes.

The read functions can take one function modifier:

- IO$M_NOW—complete the read operation immediately with a received message (if no message is currently available, return a status of SS$_ENDOFFILE in the I/O status block).

6.3.2 Write

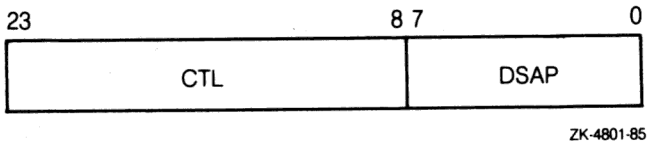
Write functions provide for the direct transfer of data from the user process's virtual memory address space to another port on the Ethernet. The VAX/VMS operating system provides three function codes:

- IO$_WRITELBLK—write logical block
- IO$_WRITEVBLK—write virtual block
- IO$_WRITEPBLK—write physical block

Transmitted DEUNA messages are copied from the requesting process's buffer to a system buffer for transmission.

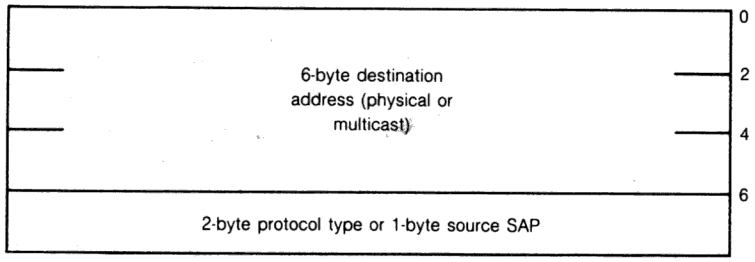
The write function takes four device/function-dependent arguments:

- P1—the starting virtual address of the buffer containing the data to be transmitted.
- P2—the size of the buffer in bytes.
- P4—a descriptor address that points to the DSAP and CTL field values (optional). (See Sections 6.1.5 and 6.1.6.) The format of the buffer is:



ZK-4801-85

- P5—the address of an eight-byte buffer that contains the destination address (either physical or multicast) and the protocol type source or SAP. If the DEUNA is in promiscuous mode, you must set either the protocol type (Ethernet format packet) in the word following the destination address, or the individual Source SAP (802 format packet) in the byte following the destination address. The individual Source SAP cannot be the NULL SAP. If the DEUNA is not in promiscuous mode, the protocol type or Source SAP (if specified) is ignored. The format of the buffer is:



The message size specified by P2 cannot be larger than 1500 bytes (see Section 6.2). (If P2 specifies a message size larger than 1500 bytes, the I/O status block returns the status SS$_IVBUFLEN.)

If the P4 buffer is specified, it must be at least three bytes long. The first byte is always the DSAP; the next two bytes are used to determine the CTL field value.

The CTL field value is either a one-byte or two-byte value. If the two least significant bits of the low-order byte of the CTL field contain "11", then just the low order byte of the CTL field is used as the CTL field value. Otherwise, both bytes of the CTL field are used as the CTL field value.

Even though the driver may only use the low-order byte of the CTL field, the user must still pass at least a three-byte buffer. In this case, the driver will use the low-order byte of the CTL field and ignore the high-order byte.

On 802 format packets, the maximum message length depends on the size of the CTL field: for a one-byte CTL field, the maximum message length is 1497 bytes; for a two-byte CTL field, the maximum message length is 1496 bytes.

If the CTL field value is one byte in length, then it is validated to be one of the three command values: UI, XID, or TEST (see Section 6.1.5).

If Class I service is enabled, only one-byte CTL field values may be passed. If user-supplied service is enabled, then both one- and two-byte CTL field values are valid. All one-byte control field values are validated to either UI, XID, or TEST; two-byte CTL field values are not validated.

The user is able to receive packets for the SAP enabled with the IO$_SETMODE or IO$_SETCHAR QIOs and to transmit packets destined for a different SAP. This would be similar to an Ethernet channel receiving packets for one protocol type and transmitting packets with a different protocol type (which is not possible with the current Ethernet $QIO interface). It is expected that most 802 format applications will only receive packets from a source SAP that matches the SAP enabled on their channel. To do this, the read function (see Section 6.3.1) has been enhanced to return the source SAP to the user. To verify that the source SAP of an incoming packet matches the SAP enabled on the channel, the user need only match the source SAP returned by the read function with the SAP enabled on the channel.

The write functions take no function modifiers.

6.3.3 Set Mode and Set Characteristics

Set mode operations are used to perform mode, operational, and program/driver interface operations with the DEUNA. The VAX/VMS operating system defines three types of set mode functions:

- Start up Ethernet port or set controller mode
- Enable attention AST
- Shut down Ethernet port

The set mode functions perform DEUNA operations, such as starting a DEUNA port and requesting an attention AST, which are described in the sections that follow. The VAX/VMS operating system provides the following two function codes, which are always used with at least one function modifier:

- IO$_SETMODE—set mode
- IO$_SETCHAR—set characteristics

6.3.3.1 Set Controller Mode

The set controller mode function sets the DEUNA controller state and characteristics, and activates the controller port. Four combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_CTRL—set controller characteristics
- IO$_SETCHAR!IO$M_CTRL—set controller characteristics
- IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP—set controller characteristics and start the controller port
- IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP—set controller characteristics and start the controller port

If the function modifier IO$M_STARTUP is specified, the DEUNA port is started. If IO$M_STARTUP is not specified, the specified characteristics are simply modified.

This function takes the following device/function-dependent argument:

- P2—the address of a descriptor for an extended characteristics buffer (optional)

The P2 buffer consists of a series of six-byte entries or counted strings. The first word contains the parameter identifier (ID) followed by either a longword that contains one of the (binary) values that can be associated with the parameter ID or a counted string. Counted strings consist of a word that contains the count followed by the character string. Figure 6–5 shows the format for this buffer.

Figure 6–5 P2 Extended Characteristics Buffer

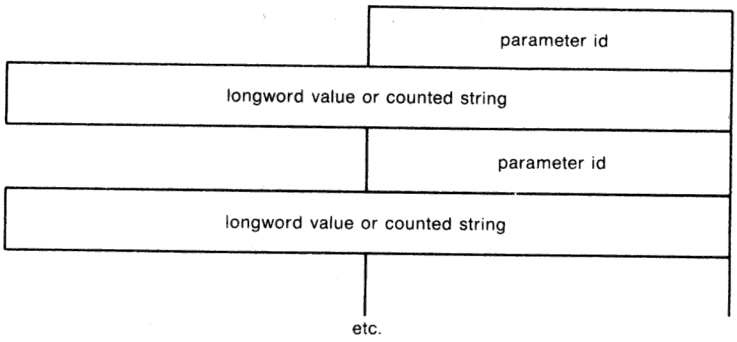


Table 6–5 lists the parameter IDs and values that can be specified in the P2 buffer. These parameter IDs are applicable to the DEUNA, the DEQNA, and the DELUA, except where otherwise noted. The $NMADEF macro defines these values. The $NMADEF macro is included in the macro library SYS$LIBRARY:LIB.MLB.

If the status SS$_BADPARAM is returned in the first word of the I/O status block, the second longword contains the code of the parameter in error.

Table 6–5 P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_ACC	Protocol access mode. This optional parameter determines the access mode for the protocol type. One of the following values can be specified:

Value	Meaning
NMA$C_ACC_EXC	Exclusive mode (default)
NMA$C_ACC_SHR	Shared default user mode
NMA$C_ACC_LIM	Shared with destination mode

Section 6.3.3.2 provides a description of protocol type sharing.

NMA$C_PCLI_ACC should not be specified on a channel where the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM_802). For this format the concept of shared protocol type is handled by using group SAPs.

NMA$C_PCLI_ACC is passed as a longword value.

NMA$C_PCLI_BFN	Number of receive buffers to preallocate. Default = 1. This optional parameter is specified on a per-protocol-type basis. It is passed as a longword value.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_BSZ	Device buffer size. This optional parameter is used by the first user of the device to change the hardware buffer size. Normally, the buffer size should not be changed from the default value (1500). The NMA$C_PCLI_BSZ parameter affects all users of the DEUNA. It is passed as a longword value.
NMA$C_PCLI_BUS	Maximum allowable receive buffer size, that is, message length (default = 512 bytes). This optional parameter is specified on a per-port basis. It is passed as a longword value.
NMA$C_PCLI_CON[1]	Controller mode. This optional parameter determines whether the DEUNA hardware is to be looped back at the DEUNA. One of the following values can be specified:

Value	Meaning
NMA$C_LINCN_NOR	Normal mode (default)
NMA$C_LINCN_LOO	Loopback mode

In loopback mode the driver always enables echo mode (NMA$C_PCLI_EKO is in the ON state). The only messages looped back are those acceptable to the DEUNA as receive messages, that is, those messages that possess at least one of the following characteristics:

- Matching physical address (see Section 6.1.2)
- Matching multicast address (see Section 6.1.2)
- Promiscuous mode (NMA$C_PCLI_PRM is in the ON state)
- Destination a multicast address and all multicasts are enabled (NMA$C_PCLI_MLT is in the ON state)

NMA$C_PCLI_CON affects all protocol types on a single DEUNA controller. It is passed as a longword value.

For the DELUA, the largest message looped is 32 bytes if CRC (NMA$C_PCLI_CRC) is enabled, or 36 bytes if CRC is disabled.

[1]If the DEUNA, DEQNA, or DELUA is active and the user does not specify this parameter, the parameter defaults to the current hardware setting. If the DEUNA, DEQNA, or DELUA is not active, this parameter will default to the default value indicated.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_CRC[2]	CRC generation state for transmitted messages. This optional parameter is applicable only to the DEUNA and DELUA device drivers. One of the following values can be specified:

Value	Meaning
NMA$C_STATE_ON	DEUNA/DELUA generates a CRC (default).
NMA$C_STATE_OFF	DEUNA/DELUA does not generate a CRC.

NMA$C_PCLI_CRC affects all protocol types on a single DEUNA or DELUA controller. There is no effect on checking a receive message's CRC. NMA$C_PCLI_CRC is passed as a longword value.

NMA$C_PCLI_CRC should not be specified on a channel where the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM_802). Disabling CRC on a channel with 802 packet format is illegal and will result in a bad parameter error.

NMA$C_PCLI_DCH — Data chaining state (optional). One of the following values can be specified:

Value	Meaning
NMA$C_STATE_ON	Allows data chaining on received messages
NMA$C_STATE_OFF	Does not allow data chaining (default)

NMA$C_PCLI_DCH affects single protocol types on a single DEUNA controller. It is passed as a longword value.

Data chaining allows the driver to receive packets in more than one receive buffer, but only if the receive buffer size is less than the maximum Ethernet size. The user process is never aware that a data chaining operation was required in the driver.

[2]If the DEUNA or DELUA is active and the user does not specify this parameter, the parameter defaults to the current hardware setting. If the DEUNA or DELUA is not active, this parameter will default to the default value indicated.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_DES	Shared protocol destination address. Passed as a counted string that consists of a modifier word (NMA$C_LINMC_SET or NMA$C_LINMC_CLR) followed by a 6-byte (48-bit) destination physical address. NMA$C_PCLI_DES only has meaning when protocol access (NMA$C_PCLI_ACC) is defined as shared with destination mode (NMA$C_ACC_LIM). NMA$C_PCLI_DES should not be specified on a channel where the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM_802). For this format the concept of shared protocol type is handled by using group SAPs. Section 6.3.3.2 provides a description of protocol type sharing.
NMA$C_PCLI_EKO[2]	Echo mode. Applicable only to the DEUNA device driver. Transmitted messages are returned to the sender. This optional parameter controls the condition of the half-duplex bit in the DEUNA mode register. One of the following values can be specified:

Value	Meaning
NMA$C_STATE_ON	Echoes transmit messages
NMA$C_STATE_OFF	Does not echo transmit messages (default)

If NMA$C_STATE_ON is specified, the only transmitted messages echoed are those acceptable to the DEUNA as receive messages, that is, those messages that have at least one of the following characteristics:

- Matching physical address (see Section 6.1.2)
- Matching multicast address (see Section 6.1.2)
- Promiscuous mode (NMA$C_PCLI_PRM) is in the ON state
- Destination a multicast address and all multicasts enabled (NMA$C_PCLI_MLT is in the ON state)

If the DEUNA is placed in loopback mode (NMA$C_LINCN_LOO is specified in the NMA$C_PCLI_CON parameter), the driver enables echo mode.

NMA$C_PCLI_EKO affects all protocol types on a single DEUNA controller.

[2]If the DEUNA or DELUA is active and the user does not specify this parameter, the parameter defaults to the current hardware setting. If the DEUNA or DELUA is not active, this parameter will default to the default value indicated.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_FMT	Packet format. This optional parameter specifies the packet format as either Ethernet or IEEE 802. The selected format must be compatible with the existing interface. This characteristic is passed as a longword value. One of the following values can be specified:

Value	Meaning
NMA$C_LINFM_ETH	Ethernet packet format (default)
NMA$C_LINFM_802	802 packet format

NMAC_PCLI_PTY, NMAC_PCLI_ACC, and NMA$C_PCLI_DES should not be specified on those channels where the 802 packet format (NMA$C_LINFM_802) is selected.

NMAC_PCLI_SRV, NMAC_PCLI_SAP, and NMA$C_PCLI_GSP should not be specified on those channels where the Ethernet packet format (NMA$C_LINFM_ETH) is selected.

NMA$C_PCLI_GSP

Group SAP. This is an optional parameter if the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM_802). If the Ethernet packet format is selected, NMA$C_PCLI_GSP cannot be specified. Group SAPs may be shared among multiple channels on the same controller. If the 802 packet format is selected, NMA$C_PCLI_GSP defines up to four 802 group SAPs that are to be enabled for matching incoming packets to complete read operations on this channel.

NMA$C_PCLI_GSP is passed as a longword value and is read as four 8-bit unsigned integers. Each integer must be either a group SAP or zero. To enable a single group SAP on a channel, the user need only specify the group SAP value to be enabled in one of the four integers and place a value of zero in the three remaining integers. To disable group SAPs on the channel, the user need only place a value of zero in all four integers.

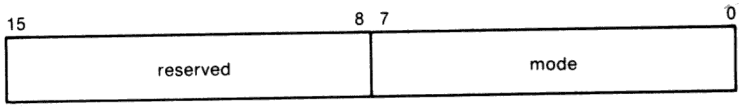
If this characteristic is correctly specified, any group SAPs that were previously enabled are now replaced by the SAPs specified by the current IO$_SETMODE or IO$_SETCHAR function.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_ILP	Internal loopback mode. This optional parameter places the DELUA in internal loopback mode (not for the DEUNA or DEQNA device drivers). One of the following values can be specified:

Value	Meaning
NMA$C_STATE_ON	Internal loopback mode
NMA$C_STATE_OFF	Not in internal loopback mode (default)

NMA$C_PCLI_MCA	Multicast address (optional). Passed as a counted string that consists of a modifier word followed by a list of 6-byte (48-bit) logical addresses. The value specified in the modifier word determines whether the addresses are set or cleared. (If NMA$C_LINMC_CAL is specified, all logical addresses in the list are ignored.) The modifier word has the following format:



ZK-1125-82

The following mode values can be specified:

Value	Meaning
NMA$C_LINMC_SET	Set the string value.
NMA$C_LINMC_CLR	Clear the string value.
NMA$C_LINMC_CAL	Clear all multicast addresses.

The driver filters all multicast addresses on a per-protocol-type basis. Therefore, only addresses enabled by the protocol will be delivered.

NMA$C_PCLI_MCA is specified on a per-protocol-type basis.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_MLT	Multicast address state. This optional parameter instructs the DEUNA hardware whether to accept multicast addresses. One of the following values can be specified:

Value	Meaning
NMA$C_STATE_ON	Accept all multicast addresses.
NMA$C_STATE_OFF	Do not accept all multicast addresses (default).

Only one protocol type may be active with multicast address mode enabled.

The NMA$C_PCLI_MLT parameter is passed as a longword value.

NMA$C_PCLI_PAD

Padding on transmit messages (optional). One of the following values can be specified:

Value	Meaning
NMA$C_STATE_ON	Padding required on short messages (default)
NMA$C_STATE_OFF	Padding not required

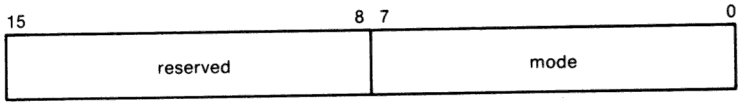
The driver verifies each transmit request to determine that at least 46 bytes of transmit data are sent. The hardware is set to always perform padding on transmit messages and never produces a short message. NMA$C_PCLI_PAD affects only the protocol type that issued the set mode request. It is passed as a longword value.

If padding is enabled on Ethernet format packets, the driver adds a 2-byte count field to the transmitted data. This allows short packets, that is, packets fewer than 46 bytes long, to be received by the receiver with the proper length returned by the driver. The minimum Ethernet packet is 46 bytes of user data. If fewer than 46 bytes were sent, then the hardware would pad the data and the receiver would always receive packets greater than 45 bytes. When padding is enabled, the maximum message size for transmit or receive operations is 1498 bytes.

Users should note that this is not the padding described in the *DEUNA User's Guide*.

NMA$C_PCLI_PAD should not be specified on a channel where the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM_802). Disabling padding on a channel with the 802 packet format is illegal and will result in a bad parameter error.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_PHA	Physical port address (optional). It is passed as a counted string that consists of a modifier word followed by the 48-bit physical address. If the request is to clear the physical port address or to set the physical port address to the DECnet default address, then the physical address (if present) is not read. The modifier word has the following format:  ZK-1125-82 One of the following mode values can be specified:

Value	Meaning
NMA$C_LINMC_SET	Set the string value.
NMA$C_LINMC_CLR	Clear the physical address.
NMA$C_LINMC_SDF	Set the physical port address to the DECnet default address. The DECnet default address is constructed by appending the low-order word of the SYSGEN parameter SCSSYSTEMID to the constant DECnet header (AA-00-04-00).

The default is the current address set by a previous set mode function on this controller, or the default hardware address if no address was defined by a previous set mode function.

The physical address must be passed as a 6-byte (48-bit) quantity. The first byte is the least significant byte. A return value of -1 on a sense mode request implies that a physical address is not defined and that the default physical address is in use.

The NMA$C_PCLI_PHA parameter affects all protocol types on a single DEUNA controller.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_PRM	Promiscuous mode (optional). One of the following values can be specified:

Value	Meaning
NMA$C_STATE_ON	Promiscuous mode enabled
NMA$C_STATE_OFF	Promiscuous mode disabled (default)

Only one unit on a DEUNA may be active with promiscuous mode specified. Enabling promiscuous mode requires PHY_IO privilege.

The NMA$C_PCLI_PRM parameter is passed as a longword value.

NMA$C_PCLI_PTY	Protocol type. This value is read as a 16-bit unsigned integer and must be different from other protocol types running on the same controller except when the protocol type is being shared. For Ethernet format channels, this required parameter is specified on a per-UCB basis; there is a UCB associated with every protocol type. NMA$C_PCLI_PTY, although a required parameter, is not used if this unit has promiscuous mode (NMA$C_PCLI_PRM) enabled, nor should NMA$C_PCLI_PTY be specified on a channel where the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM_802). NMA$C_PCLI_PTY is passed as a longword value.

Table 6–5 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_SAP	802 format SAP. This parameter is required if the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM). If the Ethernet packet format is selected, NMA$C_PCLI_SAP cannot be specified. If the packet format is set to 802, then NMA$C_PCLI_SAP defines an 802 SAP and is read as an eight-bit unsigned integer. The least significant bit of the SAP must be zero; the SAP cannot be the null SAP (all eight bits equal zero). This characteristic is passed as a longword value (only the low-order byte being used). The SAP specified by NMA$C_PCLI_SAP is the SAP used to match incoming packets to complete read requests. It is used as the source SAP (SSAP) in all transmissions (write QIOs). Because it is illegal to transmit using a group SAP as the source SAP, the SAP specified by this NMA$C_PCLI_SAP cannot be a group SAP. NMA$C_PCLI_GSP describes how to set up group SAPs on a channel. All individual SAPs specified on a controller must be unique on that controller. Therefore, the SAP specified using the NMA$C_PCLI_SAP characteristic will be checked for uniqueness on the controller. The Ethernet concept of a shared protocol type is accomplished on an 802 channel by setting up a group SAP on the channels that need to share a SAP. Group SAPs may be shared among multiple channels on the same controller.
NMA$C_PCLI_SRV	Driver service. This optional parameter specifies the service supplied by the driver. It can only be specified if the 802 packet format is selected (NMA$C_PCLI_FMT is set to NMA$C_LINFM_802). This characteristic is passed as a longword value. One of the following values can be specified:

Value	Meaning
NMA$C_LINSR_USR	User-supplied service (default)
NMA$C_LINSR_CLI	Class I service

6.3.3.2 Protocol Type Sharing

Protocol types are usually nonshareable. The problems inherent in sharing a protocol type include the multiplexing and demultiplexing of messages to and from remote nodes, and the ability to change the characteristics of a protocol type. However, the protocol access parameter (NMA$C_PCLI_ACC) allows a protocol type to be opened in either of two shareable modes: shared default (NMA$C_ACC_SHR) and shared with destination (NMA$C_ACC_LIM). The DEUNA driver also provides the nonshareable exclusive mode (NMA$C_ACC_EXC). (See Table 6–5.) The following paragraphs describe the rules and requirements for each mode:

- The exclusive mode is the default if no access mode is supplied as a P2 buffer parameter. This mode of operation does not allow the protocol to be shared by other users. Any attempt to start up another protocol of the same type results in an error status of SS$_BADPARAM.
- The shared default mode is the default user of a shared protocol type. If no other user has any protocol type/destination address association, then all receive messages go to the default user. This feature allows for other users to transmit to a particular node while sharing the common protocol type. However, there cannot be more than one default user of a protocol type.

  The protocol type must have been specified in the same (or in a previous) P2 buffer list that specified the mode. The mode cannot be changed once it has been defined as shared default.
- The shared with destination mode is a protocol type/destination address pairing that allows multiple users to share a protocol type, allowing each to communicate with different nodes. This mechanism allows the driver enough context to multiplex and demultiplex messages destined to or received from remote nodes on the Ethernet.

  The protocol type and destination address must have been specified in the same (or in a previous) P2 buffer list that specified the mode. That mode cannot be changed once it has been defined as shared with destination.

Note: If there is no shared default user of a protocol type, then incoming messages from nodes not among the "shared with destination" users for that protocol type are ignored.

6.3.3.3 Shutdown Controller

The shutdown controller function shuts down the Ethernet port. On completion of a shutdown request all buffers are returned. This port cannot be used again until another startup request has been issued (see Section 6.3.3.1).

Two combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN—shut down Ethernet port
- IO$_SETCHAR!IO$M_CTRL!IO$M_SHUTDOWN—shut down Ethernet port

The shutdown controller function takes no device/function-dependent arguments.

The driver aborts all pending I/O requests for the port on receipt of the shutdown controller request.

6.3.3.4 Enable Attention AST

This function requests that an attention AST be delivered to the requesting process when a status change occurs on the assigned protocol. An AST is queued when the driver sets or clears either an error summary bit or any of the unit status bits (see Tables 6–2 and 6–3), or when a message is available and there is no waiting read request. The enable attention AST function is legal at any time, regardless of the condition of the unit status bits.

Two combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_ATTNAST—enable attention AST
- IO$_SETCHAR!IO$M_ATTNAST—enable attention AST

This function takes the following device/function-dependent arguments:

- P1—the address of an AST service routine or 0 for disable
- P2—(ignored)
- P3—access mode to deliver AST

The enable attention AST function enables an attention AST to be delivered to the requesting process once only. After the AST occurs, it must be explicitly reenabled by the function before the AST can occur again. The function is subject to AST quotas.

The AST service routine is called with an argument list. The first argument is the current value of the second longword of the I/O status block (see Section 6.4). The access mode specified by P3 is maximized with the requester's access block.

6.3.4 Sense Mode and Sense Characteristics

The sense mode function returns the DEUNA device characteristics in the specified buffer(s). These characteristics include the device characteristics described in Section 6.2 and, with the exceptions noted below, the extended characteristics listed in Table 6–5.

Two combinations of function code and modifier are provided:

- IO$_SENSEMODE!IO$M_CTRL—read DEUNA characteristics
- IO$_SENSECHAR!IO$M_CTRL—read DEUNA characteristics

These functions take the following device/function-dependent arguments:

- P1—the address of a two-longword buffer where the device characteristics are stored. (Figure 6–6 shows the format for, and Section 6.2 describes the contents of, the P1 buffer.) The P1 argument is optional.
- P2—the address of a descriptor where the extended characteristics buffer is stored. The P2 argument is optional. The format of the buffer specified by P2 depends on whether a longword of value or a counted string is returned, as shown in Figure 6–7. The parameter ID for the buffer contains a string indicator bit (bit 12) that describes whether the data item is a string or a longword.

Except for two differences, P2 returns the same extended characteristics as those listed in Table 6–5:

- The sense mode P2 buffer does not return the modifier word for the NMA$C_PCLI_PHA and NMA$C_PCLI_MCA parameter IDs.
- In addition to the parameter IDs listed in Table 6–5, the sense mode P2 buffer returns the following parameter ID:

Parameter ID	Meaning
NMA$C_PCLI_HWA	Default physical address. Describes the value for the hardware set physical address. Read only. NMA$C_PCLI_HWA is returned in the same format as NMA$C_PCLI_PHA (see Table 6–5).

Figure 6–6 Sense Mode P1 Characteristics Buffer

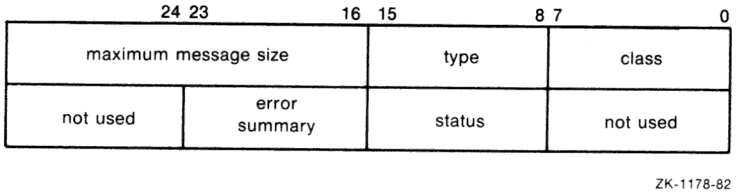
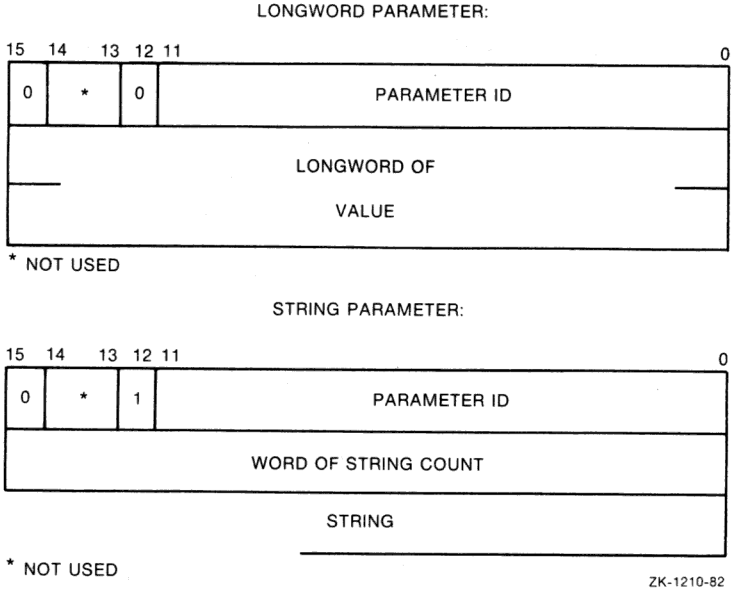


Figure 6–7 Sense Mode P2 Extended Characteristics Buffer



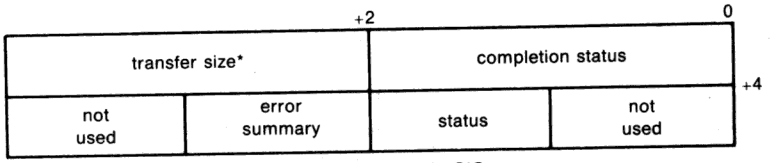
For the DEUNA and the DELUA, the minimum size that should be used for the P2 buffer is 120 bytes. Users should note that this value may change with the addition of new functionality. All characteristics that fit into the buffer specified by P2 are returned. However, if all the characteristics cannot be stored in the buffer, the I/O status block returns the status SS$_BUFFEROVF.

The second word of the I/O status block returns the size (in bytes) of the extended characteristics buffer returned by P2 (see Section 6.4).

6.4 I/O Status Block

The I/O status block (IOSB) for all DEUNA functions is shown in Figure 6–8. Appendix A lists the completion status returns for these functions. (The *VAX/VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.)

Figure 6–8 IOSB Contents



*number of bytes returned in P2 buffer if set mode QIO

ZK-1179-82

The first longword of the IOSB returns, in addition to the completion status, either the size (in bytes) of the data transfer or the size (in bytes) of the extended characteristics buffer (P2) returned by a sense mode function. The second longword returns the unit and line status bits listed in Table 6–2 and the error summary bits listed in Table 6–3.

6.5 Programming Example

This sample program (Example 6–1) shows the typical use of QIO functions in driver (both XEDRIVER and XQDRIVER) operations such as establishing the protocol type, starting the DEUNA, and transmitting and receiving data. This program does not illustrate DECnet operations because it is intended to show only basic QIO functions. The program sets the hardware in loopback mode and uses the DEUNA as a standalone device.

Example 6–1 DEUNA Program Example

```
        .TITLE      EXAMPLE         XE/XQ SAMPLE TEST PROGRAM
        .IDENT      /X01/
;++
; ABSTRACT:
;
;       This is a sample program for use with the XEDRIVER
;       (DEUNA and DELUA devices) or XQDRIVER (DEQNA device).
;--
        .LIBRARY  "SYS$LIBRARY:LIB.MLB"
        $IODEF                  ; Define I/O functions and
                                ; modifiers
        $NMADEF                 ; Define Network Management
                                ; parameters
        $XMDEF                  ; Define DMC interface parameters
        RCVBUFLEN = 512         ; Size of receive buffer
IOSB:   .BLKQ   1               ; I/O status block
RCVBUF: .BLKB   RCVBUFLEN+4     ; Enough room for buffer + CRC
XMTBUF:                         ; Start of xmit data
SETPARM:
        .WORD   NMA$C_PCLI_BUS      ; Buffer size
        .LONG   RCVBUFLEN
        .WORD   NMA$C_PCLI_BFN      ; Number of buffers
        .LONG   7
        .WORD   NMA$C_PCLI_PHA      ; Define the physical address
        .WORD   20$-10$             ; Size of the physical address
                                    ; string
10$:    .WORD   NMA$C_LINMC_SET     ; Modifier word
        .BYTE   ^XAA                ; Physical address
        .BYTE   ^X00                ; AA-00-03-00-FC-11
        .BYTE   ^X03
        .BYTE   ^X00
        .BYTE   ^XFC
        .BYTE   ^X11
20$:
        .WORD   NMA$C_PCLI_PAD      ; Pad short buffers
        .LONG   NMA$C_STATE_ON
        .WORD   NMA$C_PCLI_PTY      ; Protocol type 60-06
        .LONG   ^X0660
        .WORD   NMA$C_PCLI_PRM      ; Promiscuous mode disabled
        .LONG   NMA$C_STATE_OFF
        .WORD   NMA$C_PCLI_DCH      ; Data chaining off
        .LONG   NMA$C_STATE_OFF
        .WORD   NMA$C_PCLI_CRC      ; Generate CRC on transmit (only
        .LONG   NMA$C_STATE_ON      ; for DEUNA and DELUA)
        .WORD   NMA$C_PCLI_CON      ; Controller mode:
        .LONG   NMA$C_LINCN_LOO     ;       Loopback
SETPARMLEN=.-SETPARM
SETPARMDSC:
        .LONG           SETPARMLEN
        .ADDRESS        SETPARM
```

(Continued on next page)

Example 6–1 (Cont.) DEUNA Program Example

```
DEVDSC: .ASCID  'XEA0'              ; Unit to use for test
DEVCHAN:.BLKL   1                   ; Returned channel for I/O
                                    ; operations
DEST:   .BYTE   ^XAA                ; Destination address
        .BYTE   ^X00                ; (physical):
        .BYTE   ^X03                ; AA-00-03-00-FC-11
        .BYTE   ^X00
        .BYTE   ^XFC
        .BYTE   ^X11
;*********************************************************************
;
;               Start of code
;
;*********************************************************************
ERROR:  BRW     EXIT                ; Exit on error
;*********************************************************************
;
;       Main entry point
;
;*********************************************************************
        .ENTRY  START,^M<>
;
; Assign unit
;
        $ASSIGN_S DEVNAM = DEVDSC, CHAN = DEVCHAN
        BLBC    R0,ERROR            ; Br if error
;
; Set function to establish the protocol type and start device.
; The initial startup will take about 6 seconds for the DEUNA
; to run the self-test sequence.
;
        $QIOW_S FUNC = #<IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP>,-
                CHAN = DEVCHAN,-
                IOSB = IOSB,-
                P2 = #SETPARMDSC
        BLBC    R0,ERROR            ; Br if error
        MOVL    IOSB, R0            ; Else, get I/O status return
        BLBC    R0,ERROR            ; Br if the I/O failed
;
; Loopback data
;
        MOVZWL  #100,R9             ; Loop device 100 times
;
; Transmit some data
;
10$:    $QIOW_S FUNC=#IO$_WRITEVBLK,CHAN=DEVCHAN,-
                P1=XMTBUF,P2=#XMTBUFLEN,P5=#DEST,IOSB=IOSB
        BLBC    R0,40$              ; Br if error
        MOVL    IOSB,R0             ; Else, get I/O status return
        BLBC    R0,40$              ; Br if the I/O failed
```

(Continued on next page)

Example 6–1 (Cont.) DEUNA Program Example

```
;
; Receive the data
;
        $QIOW_S FUNC=#IO$_READVBLK,CHAN=DEVCHAN,-
                P1=RCVBUF,P2=#RCVBUFLEN,IOSB=IOSB
        BLBC    R0,40$                  ; Br if error
        MOVL    IOSB,R0                 ; Else, get I/O status return
        BLBC    R0,40$                  ; Br if the I/O failed
        SOBGTR  R9,10$                  ; Br if more to loop
;
; Shutdown the device
;
        $QIOW_S FUNC = #<IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN>,-
                CHAN = DEVCHAN,-
                IOSB = IOSB
        BLBC    R0,40$                  ; Br if error
        MOVL    IOSB,R0                 ; Else, get I/O status return
        BLBC    R0,40$                  ; Br if the I/O failed
;
;        Deassign our channel
;
        $DASSGN_S       CHAN = DEVCHAN
;
;       Exit
;
40$:
EXIT:   RET
        XMTBUFLEN= .-XMTBUF             ; Define size of transmit data
        ASSUME  XMTBUFLEN LE RCVBUFLEN
        .END    START
```

A I/O Function Codes

This appendix lists the function codes and function modifiers defined in the $IODEF macro. The arguments for these functions are also listed.

A.1 DMC11/DMR11 Interface Driver

Functions	Arguments	Modifiers
IO$_READLBLK IO$_READVBLK IO$_READPBLK	P1 - buffer address P2 - message size	IO$M_DSABLMBX IO$M_NOW
IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1 - buffer address P2 - message size	IO$M_ ENABLMBX[1]
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address	
IO$_SETMODE!IO$M_ATTNAST IO$_SETMODE!IO$M_ATTNAST	P1 - AST service routine address P2 - (ignored) P3 - AST access mode	
IO$_SETMODE!IO$M_ SHUTDOWN IO$_SETCHAR!IO$M_ SHUTDOWN	P1 - characteristics block address	
IO$_SETMODE!IO$M_STARTUP IO$_SETCHAR!IO$M_STARTUP	P1 - characteristics block address P2 - (ignored) P3 - receive message blocks	

[1]Only for IO$_WRITELBLK and IO$_WRITEPBLK

QIO Status Returns		
SS$_ABORT	SS$_BADPARAM	SS$_DATAOVERUN
SS$_DEVACTIVE	SS$_DEVOFFLINE	SS$_ENDOFFILE
SS$_NORMAL		

A.2 DMP11, DMF32, and Asynchronous DDCMP Interface Drivers

Functions	Arguments
IO$_READLBLK[!IO$M_NOW] IO$_READVBLK[!IO$M_NOW] IO$_READPBLK[!IO$M_NOW] IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size P6 - diagnostic buffer address (optional)
IO$_SETMODE IO$_SETCHAR IO$_SETMODE!IO$M_STARTUP IO$_SETCHAR!IO$M_STARTUP IO$_SETMODE!IO$M_CTRL IO$_SETCHAR!IO$M_CTRL IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP IO$_SETMODE!IO$M_SHUTDOWN IO$_SETCHAR!IO$M_SHUTDOWN IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN IO$_SETCHAR!IO$M_CTRL!IO$M_SHUTDOWN	P1 - characteristics buffer address (optional) P2 - extended characteristics buffer descriptor address (optional) P3 - receive message blocks (optional) P6 - diagnostic buffer address (optional)
IO$_SETMODE!IO$M_ATTNAST IO$_SETCHAR!IO$M_ATTNAST	P1 - AST service routine address P2 - (ignored) P3 - access mode to deliver AST
IO$_SETMODE!IO$M_SET_MODEM[1] IO$_SETCHAR!IO$M_SET_MODEM[1] IO$_SENSEMODE!IO$M_RD_MODEM[2] IO$_SENSEMODE!IO$M_RD_COUNT[1] IO$_SENSEMODE!IO$M_CLR_COUNT	P1 - modem status buffer address
IO$_SENSEMODE IO$_SENSEMODE!IO$M_CTRL	P1 - characteristics buffer address (optional) P2 - extended characteristics buffer descriptor address (optional)
IO$_SENSEMODE!IO$M_RD_MEM[1] IO$_SENSEMODE!IO$M_RD_MEM!IO$M_CTRL[1]	P1 - status slot buffer address P2 - tributary status slot address
IO$_CLEAN	(none)

[1]Only for DMP11

[2]Not for asynchronous DDCMP

QIO Status Returns		
SS$_ABORT	SS$_BADPARAM	SS$_BUFFEROVF
SS$_CANCEL	SS$_DEVACTIVE	SS$_DEVICEFULL
SS$_DEVINACT	SS$_DEVOFFLINE	SS$_ENDOFFILE
SS$_NORMAL		

A.3 DR11–W/DRV11-WA Interface Driver

Functions	Arguments	Modifiers
IO$_READLBLK IO$_READVBLK IO$_READPBLK IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size P3 - timeout period P4 - CSR value P5 - ODR value	IO$M_SETFNCT IO$M_WORD[1] IO$M_TIMED IO$M_CYCLE IO$M_RESET
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address P3 - access mode	IO$M_ATTNAST IO$M_DATAPATH[2]

[1]Not applicable to DRV11-WA

[2]Only for IO$_SETCHAR

QIO Status Returns		
SS$_BADPARAM	SS$_CANCEL	SS$_CTRLERR
SS$_DEVACTIVE	SS$_DRVERR	SS$_EXQUOTA
SS$_NOPRIV	SS$_NORMAL	SS$_OPINCOMPL
SS$_PARITY	SS$_TIMEOUT	

A.4 DR32 Interface Driver

Functions	Arguments	Modifiers
IO$_LOADMCODE	P1 - starting address of microcode to be loaded P2 - load byte count	
IO$_STARTDATA	P1 - starting address of data transfer command table P2 - length of the data transfer command table	IO$M_SETEVF

High-Level Language Subroutines	Functions
XF$SETUP	Defines command and buffer areas; initializes queues
XF$STARTDEV	Issues a request that starts the DR32
XF$FREESET	Releases command packets onto FREEQ
XF$PKTBLD	Builds command packets; releases them onto INPTQ
XF$GETPKT	Removes a command packet from TERMQ
XF$CLEANUP	Deassigns the device channel and deallocates the command area

QIO Status Returns		
SS$_ABORT	SS$_BADPARAM	SS$_BADQUEHDR
SS$_BUFNOTALIGN	SS$_CANCEL	SS$_CTRLERR
SS$_DEVACTIVE	SS$_DEVREQERR	SS$_EXQUOTA
SS$_INSFMEM	SS$_IVBUFLEN	SS$_MCNOTVALID
SS$_NORMAL	SS$_PARITY	SS$_POWERFAIL

A.5 DUP11 Interface Driver

Functions	Arguments	Modifiers
IO$_READLBLK IO$_READPBLK IO$_WRITELBLK IO$_WRITEPBLK	P1 - buffer address P2 - byte count	IO$K_SRRUNOUT IO$K_PTPBSC IO$M_LASTBLOCK[1]
IO$_SETMODE	P1 - line parameters block	IO$M_STARTUP IO$M_NODSRWAIT[2] IO$M_SHUTDOWN
IO$_SENSEMODE	(none)	

[1]Only for write functions with IO$K_PTPBSC

[2]Use only with IO$M_STARTUP

QIO Status Returns		
SS$_ABORT	SS$_ACCVIO	SS$_CANCEL
SS$_EXQUOTA	SS$_INSFMEM	SS$_NORMAL

A.6 DEUNA/DEQNA/DELUA Device Driver

Functions	Arguments	Modifiers
IO$_READLBLK IO$_READVBLK IO$_READPBLK IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size P4 - 802 format fields (optional)[2] P5 - destination address (optional)[2]	IO$M_NOW[1]
IO$_SETMODE IO$_SETCHAR	P2 - extended characteristics buffer (optional)[3]	IO$M_CTRL IO$M_STARTUP IO$M_SHUTDOWN
IO$_SETMODE IO$_SETCHAR	P1 - AST service address P3 - access mode to deliver AST	IO$M_ATTNAST
IO$_SENSEMODE	P1 - device characteristics buffer (optional) P2 - extended characteristics buffer (optional)	IO$M_CTRL

[1]Only for read functions.

[2]See text for complete contents.

[3]Use only with IO$M_CTRL alone or with IO$_STARTUP, that is, the set controller mode.

QIO Status Returns		
SS$_ABORT	SS$_ACCVIO	SS$_BADPARAM
SS$_BUFFEROVF	SS$_COMMHARD	SS$_CTRLERR
SS$_DATACHECK	SS$_DATAOVERUN	SS$_DEVACTIVE
SS$_DEVALLOC	SS$_DEVINACT	SS$_DEVOFFLINE
SS$_DEVREQERR	SS$_DISCONNECT	SS$_DUPUNIT
SS$_ENDOFFILE	SS$_EXQUOTA	SS$_INSFMEM
SS$_INSFMAPREG	SS$_IVBUFLEN	SS$_MEDOFL
SS$_NOPRIV	SS$_NORMAL	SS$_OPINCOMPL
SS$_TIMEOUT	SS$_TOOMUCHDATA	

Index

A

B

C

D

E

F

I

M

P

Q

S

V

X

VAX/VMS
I/O User's Reference
Manual: Part II
AA–Z601B–TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line

VAX/VMS
I/O User's Reference
Manual: Part II
AA–Z601B–TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line

NOTES

NOTES

VAX/VMS DELTA/XDELTA Utility Reference Manual

Order Number: AA–Z412A–TE

September 1984

This manual describes the VAX/VMS DELTA/XDELTA Utility.

Revision/Update Information:	This is a new manual.
Software Version:	VAX/VMS Version 4.0

digital equipment corporation
maynard, massachusetts

September 1984

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital

ZK–2305

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

DELTA/XDELTA Contents

Preface

Intended Audience

This document is for those people who must debug system code, especially device drivers and other images that execute in privileged processor-access modes or at an elevated IPL.

Conventions Used in This Document

Convention	Meaning
RET	A symbol with a one- to three-character abbreviation indicates that you press a key on the terminal, for example, RET .
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec,...	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks (″). The term apostrophe (′) is used to ref to a single quotation mark.

Structure of This Document

This document is composed of three major sections.

The Format Section is an overview of DELTA and XDELTA, and is intended as a quick reference guide. The format summary contains the DCL commands that invoke DELTA, listing all qualifiers and parameters. The usage summary describes how to invoke and exit from the DELTA and XDELTA, and any restrictions you should be aware of. The command summary lists all DELTA's and XDELTA's commands.

The Description Section explains how to use DELTA and XDELTA.

The Commands Section describes each of DELTA's and XDELTA's commands. The commands appear in alphabetical order.

DELTA/XDELTA

DELTA and XDELTA are debugging tools that you can use to monitor the execution of user programs and the VAX/VMS operating system. They are documented together here because most of their commands are identical.

You can use DELTA to debug user-mode programs or programs that execute at IPL 0 in any processor mode. You can use XDELTA to debug programs that execute in any processor mode and at any IPL. You cannot use DELTA to debug code that executes at an elevated IPL.

FORMAT

$ RUN SYS$LIBRARY:DELTA

Command Qualifiers
None.

Defaults
None.

Command Parameters

None.

usage summary

Invoking
To use DELTA to debug a program, link the program with the /DEBUG qualifier. Then define the logical name LIB$DEBUG to be SYS$LIBRARY:DELTA.EXE. Then use the RUN command to run the program.

To debug code that executes at an elevated IPL, including driver code and the executive, use XDELTA. You should use XDELTA on a stand-alone system because it interferes with normal timesharing operations.

Exiting
To exit from DELTA, type the EXIT command.

To exit from XDELTA, use the ;P command to proceed from the breakpoint.

Directing Output
You cannot redirect output from DELTA or XDELTA; but, because you use XDELTA at the console terminal, you will have a written record of your activity when you use XDELTA.

Privileges/Restrictions
To run DELTA in a processor mode other than user mode, your process must have the privilege that allows DELTA to change to that mode: change-mode-to-executive (CMEXEC) or change-mode-to-kernel (CMKRNL) privilege. Furthermore, when DELTA encounters a breakpoint, it gets control in the access mode prevailing when it encountered the breakpoint.

Furthermore, to use the ;M command in DELTA, your process must have change-mode-to-kernel (CMKRNL) privilege.

To use XDELTA, you must be able to boot the system.

commands

Syntax
[address-expression] command [expression]

DELTA/XDELTA Commands
RETURN (Close Current Location)
LINEFEED (Close Current Location and Open Next)
ESC (Open and Display Previous Location)
= (Display Value of Expression)
;E (Execute Command String)
;G (Go)
/ (Open Location and Display Contents in Prevailing Mode)
! (Open Location and Display Contents in Instruction Mode)
TAB (Open Location Specified by Q, the Current Value)
;P (Proceed from Breakpoint)
" (Set ASCII Mode)
;X (Set Base Register)
;B (Set, Clear, or Display Breakpoint)
[(Set Display Mode)
S (Step Instruction)
O (Step Instruction Over Subroutine)
;M (Set All Processes Writeable)
'string' (Deposit ASCII String)

DESCRIPTION

Both DELTA and XDELTA have the same commands, use the same expressions, and can be used to debug programs. But they are different: you use them to debug different kinds of code, and you invoke and exit from them in different ways.

The next few sections describe these differences. Later sections describe the DELTA and XDELTA commands, which are, for the most part, identical.

1 Using DELTA

You can use DELTA to debug programs that execute at IPL 0. To debug a program that executes in user mode, you need no privileges. To debug a program that executes in another processor-access mode, the process in which you link your program must have the necessary privileges. If your program executes in executive mode, for example, your process must have change-mode-to-executive (CMEXEC) privilege.

1.1 Invoking DELTA

To invoke DELTA, give your process the privileges needed to perform the change-mode instructions the program contains. Then link your program with DELTA, and use the RUN command to execute your program.

The following commands link DELTA with your program and start the debugging process:

```
$ LINK program-name
$ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA
$ RUN/DEBUG program-name
```

When DELTA begins execution, it displays its name and its current version number, like this:

```
DELTA Version 2.3
```

DELTA then executes a breakpoint on the first instruction in the program with which it is linked. It displays the address of that instruction, a slash (/), and the instruction and its operands. DELTA is then ready for your commands.

1.2 Exiting from DELTA

To exit from DELTA, type EXIT.

If you have defined a symbol named EXIT in your program, DELTA displays the value of this symbol. To exit, you must type the EXIT command once more.

2 Using XDELTA

You can use XDELTA to debug code that executes at elevated IPL. This code includes VAX/VMS executive routines, device drivers, and other privileged code.

Because XDELTA breakpoints interrupt program execution at IPL 31, no other system activity can take place. For this reason you should use XDELTA only on a stand-alone system.

2.1 Invoking XDELTA

To use XDELTA, you must bootstrap the system using a command procedure that includes XDELTA in the system. You must then request an XDELTA interrupt.

The procedures for bootstrapping the system with XDELTA differ depending on the processor on which the system is being booted. Each one, however, causes XDELTA to be included in the system, and causes XDELTA to place a breakpoint in the VAX/VMS initialization routine. Execution of the breakpoint instruction transfers program control to a fault handler located in XDELTA.

2.1.1 Bootstrapping the System on a VAX–11/780

In addition to the normal system bootstrap command files, the VAX/VMS console floppy diskette for a VAX–11/780 contains three command files that bootstrap the system with XDELTA:

- DUAXDT
- DMAXDT
- DBAXDT

To bootstrap the system with XDELTA, follow the procedures in the *Guide to VAX/VMS System Management and Daily Operations* with two exceptions:

- In R3, deposit the number of the device from which you want to boot the system.
- Specify one of the command files listed above rather than one of the command files listed in the *Guide to VAX/VMS System Management and Daily Operations*.

For example, to bootstrap the system with XDELTA on a VAX–11/780, type the following command:

```
>>> DEPOSIT R3 0
```

This command deposits zero, the number of the unit from which to boot the system, in R3.

To invoke the command procedure that boots the system from DMA0, type the following command:

```
>>> @DMAXDT
```

The procedure boots the processor and prompts you from SYSBOOT. When the **SYSBOOT>** prompt appears, you can enter any SYSBOOT command. If you do not set or load system parameters with the USE command, the system uses the parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, you can set the system parameter BUGREBOOT to 0.

To continue the bootstrapping operation, type the CONTINUE command.

2.1.2 Bootstrapping the System with XDELTA on a VAX–11/750 or MicroVAX I

To bootstrap VAX/VMS with XDELTA on a VAX–11/750 or a MicroVAX I, issue the following command:

```
>>> B/n device-name
```

The **B** is the console's BOOT command.

The **device-name** is the name of the device from which to bootstrap the system. Specify the device-name using the format ddcu. (See the *Guide to VAX/VMS System Management and Daily Operations* or the *MicroVMS User's Manual* for a complete description of the format of device names.) You must specify identifiers for both the controller and the unit identifiers; there are no defaults.

The /n qualifier loads the value n into R5. The contents of R5 are passed as input to VMB.EXE. The value of n must be one of the following 32-bit, hexadecimal numbers described below.

Value	Meaning
0	Normal, nonstop bootstrap (default)
1	Stop in SYSBOOT (equivalent to @DxyGEN on the VAX–11/780)
2	Include XDELTA with the system, but do not take the initial breakpoint
6	Include XDELTA with the system, and take the initial breakpoint
7	Include XDELTA with the system, stop in SYSBOOT, and take the initial breakpoint at system initialization (equivalent to @DxyXDT on the VAX–11/750)

For example, the following command bootstraps the system on a VAX–11 /780, the bootstrap device being DMA0.

```
>>> B/7 DMA0
```

The /7 qualifier includes XDELTA in the system and stops the booting process in SYSBOOT, which prompts you. It also causes XDELTA to set a breakpoint in the system initialization routine.

```
SYSBOOT>
```

You can enter SYSBOOT commands when SYSBOOT gives you the SYSBOOT> prompt. If you do not set or load system parameters with the USE command, the system uses the parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, you can set the system parameter BUGREBOOT to 0.

To continue the bootstrapping operation, type the CONTINUE command. See the *VAX–11/750 Software Installation Guide* or the *MicroVAX I User's Manual* for further details on the B command.

To bootstrap the system from the console TU58, see the *VAX–11/750 Software Installation Guide*. The console TU58 contains the command files DUAXDT, DMAXDT, and DBAXDT, which contain the command procedures that boot the system from DU, DM, and DB devices, respectively.

2.1.3 Bootstrapping the System on a VAX–11/730

In addition to the normal system bootstrap command files, the VAX/VMS console DECtape for a VAX–11/730 contains two command files that bootstrap the system with XDELTA: DQAXDT and DQ0XDT.

To bootstrap a VAX–11/730 with XDELTA, follow the procedures outlined in the *VAX–11/730 Software Installation Guide*, but specify DQAXDT or DQ0XDT. For example, to bootstrap the system with XDELTA on a VAX–11 /730 using the DQAXDT command procedure, type the following command.

```
>>> D/G/L 3 1
```

This command deposits the unit number, one, in R3. Then type:

```
>>> @DQAXDT
```

This command procedure boots the system from DQA1.

If the boot device is DQA0, you can omit these two steps, and instead execute the command procedure DQ0XDT, as shown below.

```
>>> @DQ0XDT
```

This command procedure does not require the unit number as a parameter.

Either of these procedures boots the processor and prompts you from SYSBOOT. When SYSBOOT prompts you, you can enter any SYSBOOT command. If you do not set or load any system parameters with the USE command, the system uses the system parameters stored in the system image. To prevent the system from automatically rebooting after a bugcheck, you can set the system parameter BUGREBOOT to 0.

To continue the bootstrapping process, type the following command.

```
SYSBOOT> CONTINUE
```

2.2 Requesting an Interrupt

When the system has bootstrapped itself, you need to request an XDELTA interrupt so that XDELTA gets control of the system. To request a software interrupt from which XDELTA gets control on a VAX–11/780, a VAX–11/750, or a VAX–11/730, issue the following commands at the console terminal:

```
$ CTRL/P
>>> D/I 14 5
>>> C
```

These commands request an interrupt at IPL 5, at which XDELTA gets control of the system.

If your system is a VAX–11/782, request the interrupt at IPL 12 by typing C instead of 5 in the example above. (Requesting an XDELTA interrupt at IPL 12 is also useful in those cases in which the system is hung at an IPL between 5 and 12.) Type these commands on the console of the primary processor, then again on the console of the attached processor.

2.3 Setting the Initial Breakpoint

When debugging a device driver's initialization routines, you should set a breakpoint in the driver code so that XDELTA encounters the breakpoint before any of the driver's initialization routines are called. To do this, place a call to the system routine INI$BRK in the code.

An example of such a call is the following, which calls the INI$BRK system routine as a subroutine:

```
JSB G^INI$BRK
```

You should place such a breakpoint in the driver's code because SYSGEN's CONNECT command calls the driver's controller-initialization and unit-initialization routines. To debug these routines, you must allow XDELTA to gain control of the driver's execution before these routines execute.

The INI$BRK routine contains two instructions: BPT and RSB. You can use the INI$BRK routine as a debugging tool, placing calls to this routine in any part of the source code you want to debug. After the break is taken, the return address, the address in the driver to which control returns when you proceed from the breakpoint, is on the top of the stack.

Note that INI$BRK is defined XDELTA's breakpoint 1. When using XDELTA, never clear breakpoint 1.

2.4 Proceeding from a Breakpoint

After the system has bootstrapped, it displays its welcoming message and halts in XDELTA, which displays this message:

```
1 BRK AT nnnnnnnn
address/NOP
```

Now XDELTA is waiting for input. (In most circumstances XDELTA does not prompt you.) Usually, you proceed from this point with the following command:

```
;P
```

If the system halts with a fatal bugcheck, the system prints the bugcheck information on the console terminal. Then, if the system parameter BUGREBOOT was set to 0, XDELTA prompts you. Bugcheck information consists of:

- The type of bugcheck
- The contents of the registers
- A dump of one or more stacks

The contents of the PC and the stack indicate the cause of the failure, usually an error in a user-written device driver. You can then examine the system's state further by issuing XDELTA commands.

2.5 Exiting from XDELTA

Exit from XDELTA by clearing all breakpoints but breakpoint 1 and proceeding from the breakpoint with the ;P command.

3 Using DELTA and XDELTA Commands

Commands for DELTA and XDELTA work identically. With few exceptions, DELTA and XDELTA have the same commands. DELTA has two commands that XDELTA does not: EXIT and ;M. XDELTA defines some symbols that DELTA does not. These differences are noted where these commands and symbols are discussed.

This section describes how to use DELTA and XDELTA commands to perform debugging operations. In addition, this section describes how to form symbolic expressions, arguments to many DELTA and XDELTA commands.

3.1 Symbols Supplied by DELTA and XDELTA

DELTA and XDELTA define symbols that are useful in forming expressions and referring to registers. The symbols are the following:

Symbol	Value
.	The current address, the address of the current location. The value of this symbol is set by the open-location-and-display-contents (/), the open-location-and-display-instruction (!), and the open-location-and-display-indirect (TAB) commands.
Q	The last value displayed. The value of Q is set by every command that causes DELTA or XDELTA to display the contents of memory.
Xn	Base register n, where n can range from 0 to F (hexadecimal). These registers are used for storing values, most often the base addresses of data structures in memory. XDELTA, but not DELTA, uses XE and XF to store the addresses of two command strings that XDELTA stores in memory. See the execute-command-string command for more information. XDELTA, but not DELTA, also uses registers X4 and X5 to contain useful addresses. X4 contains SCH$GL_CURPCB, the symbolic address of the location that contains the address of the PCB of the current process. X5 contains SCH$GL_PCBVEC, the symbolic address of the start of the PCB vector, the list of PCB slots.
Rn	General register n, where n can range from 0 to F (hexadecimal). Note that RF+4 is the processor status longword (PSL), and RE is the stack pointer.
Pn	The internal processor register at processor address n, where n can range from 0 to 3F (hexadecimal). See the *VAX Hardware Handbook* for a description of these processor registers.
G	^X80000000, the prefix for system space addresses. G2E, for example, is equivalent to ^X8000002E.
H	^X7FFE0000, the prefix for addresses in the control region (P1 space). H2E, for example, is equivalent to ^X7FFE002E.

3.2 Forming Numeric Expressions

Expressions are combinations of numbers, symbols that have numeric values, and arithmetic operators. XDELTA stores and displays all numbers in hexadecimal. It also interprets all numbers as hexadecimal.

Expressions are formed using regular (infix) notation, rather than Polish or reverse Polish notation. XDELTA ignores operators that trail the expression. The following is a typical expression:

```
G4A32+24-.
```

XDELTA evaluates expressions from left to right, and no operator takes precedence over any other. Trailing operators are ignored.

XDELTA recognizes five binary arithmetic operators, of which one also acts as a unary operator. They are listed below.

Operator	Action
+ or SPACE	Addition
-	Subtraction, when used as a binary operator, or negation, when used as a unary operator
*	Multiplication
%	Division
@	Arithmetic shift

The example below shows the arguments required by the arithmetic-shift operator.

```
n@j
```

In the example above, **n** is the number to be shifted, and **j** is the number of bits to shift it. If j is positive, n is shifted to the left; if j is negative, n is shifted to the right. Argument j must be less than 20 (hexadecimal) and greater than -20 (hexadecimal). Bits shifted beyond the limit of the longword are lost. Argument n must be less than or equal to FFFFFFFF (hexadecimal).

COMMANDS

The commands for DELTA and XDELTA act identically. Except for a few commands, DELTA and XDELTA have the same commands. The EXIT and Set-All-Processes-Writeable (;M) commands are only used with DELTA.

All other commands described in this section are for use with both DELTA and XDELTA, but in most cases only DELTA is mentioned. Any differences in using a command with DELTA or XDELTA, such as the fact that EXIT is used only with DELTA, are stated in the description of the command.

RETURN (Close Current Location)

Closes a location that has been opened by an open-and-display command.

FORMAT

RETURN

DESCRIPTION

Pressing the RETURN key causes one of several actions, depending on the context in which you press it. It acts as a terminator for all other commands. Also, if you have opened a location with one of the open-and-display commands, the RETURN key closes that location. Further, it acts as an ASCII character within quoted strings.

See the individual descriptions of the open-and-display commands; see the description of the deposit-ASCII-string command (') for information on quoted strings.

LINEFEED (Close Current Location, Open Next)

Closes the currently open location and opens the next location, displaying its contents.

FORMAT

LINEFEED

DESCRIPTION

The close-current-location-open-next command closes the currently open location, then opens the next and displays its contents. This command accepts no arguments, and thus can only be used to open the next location. It is useful for examining a series of locations one after another.

ESC (Open and Display Previous Location)

Opens the previous location and displays its contents.

FORMAT

ESC

DESCRIPTION

The open-and-display-previous-location command decrements the location counter (.) by the width (in bytes) of the prevailing display mode, opens that many bytes, and displays their contents. The address of the location is displayed on a new line, followed by a slash (/) and the contents of that address. The contents are displayed in the prevailing display mode.

This command is ignored if the prevailing display mode is instruction mode.

= (Display Value of Expression)

Evaluates an expression and displays its value.

FORMAT

expression =

argument

expression
The expression to be evaluated

DESCRIPTION

The display-value-of-expression command evaluates an expression and displays its value. The expression can be any valid DELTA expression. See Section 3.2 for a description of DELTA expressions.

DELTA displays the value of the expression as a hexadecimal number.

;E (Execute Command String)

Executes a string of DELTA commands stored in memory.

FORMAT

address-expression **;E**

arguments

address-expression
The address of the string of DELTA commands to execute

DESCRIPTION

The execute-command-string command executes a string of DELTA commands. The commands must be stored in memory as ASCII text. See deposit-ASCII-string command (') for information on how to store strings in memory.

If you want DELTA to proceed with program execution after it executes the string of commands, end the command string with the ;P command. If you want DELTA to wait for you to type command after it executes the string of commands, end the command string with a null byte (a byte containing 0).

XDELTA, but not DELTA, provides two command strings in memory. The addresses of these command strings are stored in XE and XF. The string addressed by XE displays the PFN database for the PFN in X0. The string addressed by XF copies the PFN in R0 to base register X0, then displays the PFN database for that PFN.

You can use these command strings to obtain the following information.

- Specified physical page number (PFN)
- PFN state
- PFN type
- PFN reference count
- PFN backward link or working-set-list index
- PFN forward link or share count
- The page table entry (PTE) that points to this PFN
- The PFN backing-store address
- Virtual block number in the process swap image, the block to which this page's entry in the SWPVBN array points

;G (Go)

Continues program execution.

FORMAT

address-expression;G

arguments

address-expression
The address at which to continue program execution

DESCRIPTION

The go command places the address specified by **address-expression** into the PC, and to continue execution of the program at that address.

EXAMPLE

```
45F80;G
```

This command places 45F80 into the PC, and then begins execution of the program at that address.

/ (Open Location and Display Contents in Prevailing Mode)

Opens a location and displays its contents in the prevailing display mode.

FORMAT

[pid:][addr-exp] old [new-exp]

arguments

pid

The internal PID of the process for which you want to display the contents of a location; if zero is specified, the process is that in which you are running DELTA.

Use of this parameter causes subsequent open-location-and-display commands to display the contents of locations in this process until another PID is specified with this command.

You can obtain the internal PID of the processes of interest by running SDA and using SDA's SHOW SUMMARY command.

addr-exp

The address of the location to be opened, or the range of addresses to be opened

old

This is actually not a parameter, but is the representation, in the prevailing display-mode, of the contents of the location or range of locations specified by the pid and addr-exp arguments.

new-exp

An expression, the value of which is to be deposited into the location. If specified, and if a process-id is also specified, you must have already set the target process to be writeable by means of the ;M command.

DESCRIPTION

The open-location-and-display-contents-in-prevailing-mode command opens the location at **address-expression** and displays **old-contents**, the contents of that location, in hexadecimal format. You can place a new value in the location by specifying an expression as shown above. If you place a new value in the location, the old contents are lost.

To display a range of locations, the address-expression parameter is the first address in the range, followed by a comma, followed by the last address in the range.

This command changes the current address (.) to the contents of the opened location. It also sets the prevailing display mode from instruction-mode to hexadecimal-mode.

Note that a subsequent close-location command does not change the current address; but a subsequent close-current-and-open-next command closes the location opened by the open-and-display-location command, adds to the address of that location the number of bytes in the prevailing display mode, makes that the current address, and then opens that location and displays its contents.

In DELTA, but not XDELTA, you can examine the address space of any existing process, provided your process has CMKRNL privilege. To do so, the format of the open-location-and-display-contents command is as follows:

```
process-id:address-expression/contents-of-location
```

The **process-id** argument is the internal PID of the process in which you want to examine locations. The other arguments are the same as those used to examine a location in your current process.

EXAMPLES

1

```
expression-1/ expression-2 / expression-3
```

The first command, **expression-1/**, opens the location addressed by expression-1 and displays the contents of that location, **expression-2**. The next open-location-and-display-contents command opens the location addressed by expression-2 and displays **expression-3**, the contents of that location.

2

```
start-addr-expression,end-addr-expression/contents-of-start-addr
second-addr/contents-of-second-addr
third-addr/contents-of-third-addr
   .
   .
   .
end-addr/contents-of-end-addr
```

This example shows the open-and-display-contents-in-prevailing-mode command used with a range of addresses. This command displays the contents of the first location, then the address and contents of each subsequent location within the range.

! (Open Location and Display Contents in Instruction Mode)

Displays an instruction and its operands.

FORMAT

address-expression !

arguments

address-expression
The address of the instruction to display

DESCRIPTION

The open-location-and-display-contents-in-instruction-mode command displays the contents of memory as a MACRO instruction, starting with the address you specify. DELTA does not make any distinction between reasonable and unreasonable instructions or instruction streams; the decoding always begins at the specified address.

This command does not allow you to modify the contents of the location. The command sets a flag that causes subsequent close-current-and-display-next or open-and-display-indirect-location commands to perform instruction decoding. You can clear the flag by using the open-location-and-display-contents command, which displays the contents of the location as a hexadecimal number.

When an address appears as an instruction's operand, DELTA sets the Q (the last-quantity-displayed variable) to that address. DELTA changes Q only for operands that use program-counter or branch-displacement addressing modes; Q is not altered for operands that use literal and register addressing modes. This feature is useful for following branches, as shown in the example below.

EXAMPLE

```
address-expression!BRW address-2!MOVL R1,R3
```

In this example, **address-expression** is the address of the first instruction to display. DELTA displays the instruction, **BRW**, and its operand, **address-2**. Because this operand does not use literal addressing or register-mode addressing, DELTA sets Q to be address-2.

The second display-instruction command uses the value of Q, the default address, as its argument. It displays MOVL, the instruction at location address-2, and its operands, R1 and R3.

TAB (Open and Display Indirect Location)

Opens the location addressed by the contents of the current location, and displays its contents.

FORMAT

TAB

DESCRIPTION

The open-and-display-indirect-location command opens the location addressed by the contents of the current location, and displays its contents, The display is made in the prevailing display mode. This command is useful for examining data structures that have been placed in a queue, or the operands of instructions.

This command changes the current address (.) that of the location displayed.

This command does not effect the display mode.

EXAMPLE

```
address-expression/address-expression-2 TAB
address-expression-2/contents-of-address-expression-2
```

The open-and-display-contents command opens the location at address-expression, and displays that location's contents, address-expression-2. The open-location-and-display-indirect command opens the location addressed by the value of Q, the last value displayed, which in this case is address-expression-2.

;P (Proceed from Breakpoint)

Causes DELTA to continue program execution following the encounter of a breakpoint.

FORMAT

;P

DESCRIPTION

The proceed-from-breakpoint command causes DELTA to continue program execution at the address contained in the program's PC.

"(Open and Display Contents in ASCII)

Displays the contents of a location as an ASCII string.

FORMAT

address-expression"

arguments

address-expression
The address of the location whose contents are to be displayed

DESCRIPTION

The open-and-display-contents-in-ASCII command causes DELTA to open the location at **address-expression** and display its contents in ASCII format.

The display mode remains ASCII until the next open-and-display-location command (/) or display-instruction command (!). These commands change the display mode to hexadecimal or instruction, respectively. The width of the location displayed, byte, word, or longword, remains unchanged by the open-location-and-display-contents-in-prevailing-mode command, but is changed by the open-location-and-display-instruction command.

EXAMPLE

```
235FC2 [W/ 415A
235FC2" ZA LINEFEED
235FC4/ PP
```

The open-location-and-display-ASCII command (") changes the prevailing display mode to ASCII, but does not affect the width of the display. The next open-and-display-next command (LINEFEED) determines the address of the location to open by adding the width, in bytes, to the value of ., then opens the number of bytes equal to the width of the prevailing display mode, which in this case is two bytes.

Note that the ASCII representation of the contents of the location presents the bytes left to right, whereas the hexadecimal representation presents them right to left.

;X (Load Base Register)

Places an address in a base register.

FORMAT

address-expression,n **;X**

arguments

address-expression
The address to place in the base register

n
The number of the base register

DESCRIPTION

To place an address in a base register, type an expression followed by a comma (,), a single digit between 0 and D (hexadecimal), a semicolon (;), and the letter X. DELTA places the specified expression in base register n. DELTA confirms that the base register is set by displaying the value deposited in the base register.

When DELTA displays an address that is within 800 (hexadecimal) bytes of an address stored in a base register, DELTA displays the base register identifier (Xn), followed by an offset that gives the address's location in relation to the address stored in the base register. If the address falls outside this range, DELTA displays it as a hexadecimal value.

If base register 2 contains 800D046A, for example, and the address an DELTA command needs to display is 800D052E, DELTA displays that address as X2+C4. DELTA computes relative addresses for both opened or displayed locations, and for addresses that are instruction operands.

DELTA and XDELTA provide symbols that make it easy to refer to processor, general, and base registers. See Section 3.1 for a description of these symbols.

;B (Breakpoint)

Shows, sets, and clears breakpoints.

FORMAT

[addr][,n][,display][,cmd] **;B**

arguments

addr

The address at which to set the breakpoint

n

The number to assign to this breakpoint, a number between 2 and 8. If omitted, DELTA assigns the first unused number to the breakpoint; if all numbers are in use, DELTA displays its only error message, "EH?".

display

The address of a location, the contents of which are to be displayed in the prevailing display mode when this breakpoint is encountered. If omitted, DELTA displays only the instruction that begins at the specified address.

cmd

The address of the string of DELTA commands to execute when this breakpoint is encountered. If omitted, DELTA executes no commands automatically, but waits for you to enter commands interactively.

DESCRIPTION

The breakpoint command shows, sets, and clears breakpoints. The action of this command depends on the arguments used with it. Each action is described below.

Displaying Breakpoints

To show all the breakpoints currently set, type **;B**. For each breakpoint, DELTA displays the following information:

- The number of the breakpoint
- The address at which the breakpoint is set
- The address of a location whose contents are to be displayed when the breakpoint is encountered
- The address of the command string associated with this breakpoint (for complex breakpoints)

Setting Breakpoints

- To set a breakpoint, type an address followed by a semicolon (;), the letter B, then press RETURN, as shown below.

```
addr-exp;B RETURN
```

DELTA sets a breakpoint at the specified location, and assigns it the first available breakpoint number.

Before DELTA executes the instruction at which a breakpoint is set, it suspends normal instruction processing, and sets a flag that causes subsequent close-and-display-next or display-indirect-location commands to perform instruction decoding. Then it displays the following message, where **n** is the number of the breakpoint.

```
n BRK at address
address/decoded-instruction
```

When the display appears, you can enter other DELTA commands. You can reset the flag that controls the mode in which instructions are displayed by issuing the open-location-and-display-contents command.

Setting a Breakpoint and Assigning a Number to It

To set a breakpoint and assign it a number, type an address followed by a comma, a single digit between 2 and 8, a semicolon (;), the letter B, and then press RETURN, as shown below.

```
addr-exp,n;B RETURN
```

DELTA sets a breakpoint at the specified location and assigns it the specified breakpoint number. Note that breakpoint 1 is reserved for INI$BRK in XDELTA, but is undefined for DELTA.

Clearing Breakpoints

To clear a breakpoint, type zero (0), followed by a comma, the number of the breakpoint to remove, a semicolon (;), the letter B, and then press RETURN. DELTA clears the specified breakpoint.

When using XDELTA, do not clear breakpoint 1. If you do, any breakpoint instructions you have coded into your driver will result in unrecognized-breakpoint exceptions.

The following shows an example of clearing breakpoint n.

```
0,n;B RETURN
```

Setting Complex Breakpoints

A complex breakpoint is one that, when encountered, performs the actions listed below.

- Displays the next instruction to be executed
- Displays the contents of another, specified location
- Executes a string of DELTA commands stored in memory

The string of commands executes after the instruction is displayed. See the deposit-ASCII-string command (') for information on storing a string of DELTA commands in memory.

To set a complex breakpoint, type the ;B command as shown below.

```
addr-exp,n,display-addr-exp,cmd-string-addr;B
```

The **addr-exp** is an expression whose value is the location at which the breakpoint is to be set.

The **n** is the number to assign to this breakpoint. The number of the breakpoint can range from 1 to 8.

The **display-addr-exp** is an expression, the value of which is the address of a location whose contents are to be displayed when this breakpoint is encountered.

The **cmd-string-addr** is an expression, the value of which is the address of the string of DELTA commands to be executed when this breakpoint is encountered. DELTA displays the information requested before executing the string of commands associated with complex breakpoints.

[(Set Display Mode)

Sets the mode of the displays produced by DELTA commands

FORMAT

[*mode*

arguments

mode

A letter that indicates the mode in which the display is made, one of the following:

Mode	Meaning
B	Byte mode. In this mode each open-and-display-location command displays the contents of one byte of memory.
I	Instruction mode. In this mode each open-and-display-location command displays the contents of memory as an instruction and its operands, displaying as many bytes of memory as necessary to display all operands.
L	Longword mode. In this mode each open-and-display-location command displays the contents of a longword of memory. This is the default mode.
W	Word mode. In this mode each open-and-display-location command displays the contents of one word of memory.

DESCRIPTION

The set-display-mode command changes the prevailing display width to byte, word, or longword. The default display width is longword. The display mode remains in effect until you give another display-mode command.

To change the prevailing display mode to instruction format, type [I. This command is equivalent to the display-instruction command, and, similarly, is canceled by typing an open-and-display-location command (/ or !) or an open-and-display-location-in-ASCII (") command.

Setting the display mode to instruction mode also sets display mode to longword, as does the set-display-mode command used with the L argument.

EXAMPLE

```
address-expression [B/ old-value new-value
```

This command displays the least significant byte contained at the address **address-expression**, and deposits **new-value** in that byte.

S (Step Instruction)

Executes one instruction and displays the next, stepping into a subroutine, if the instruction is a call to a subroutine, and displaying the next instruction to be executed.

FORMAT

S

DESCRIPTION

The step-instruction command causes DELTA to execute one instruction, display the address of the next instruction, and display that next instruction.

This command also sets a flag that causes subsequent close-and-display-next or display-indirect-location commands to perform instruction decoding. The open-location-and-display-contents command clears the flag, causing the display mode to revert to longword, hexadecimal mode.

If the next instruction is BSBB, BSBW, JSB, CALLG, or CALLS, this command executes that instruction and displays the first instruction within the subroutine.

O (Step Instruction Over Subroutine)

Executes one instruction and displays the next, stepping over a subroutine by executing it and displaying the instruction to which the subroutine returns control.

FORMAT

O

DESCRIPTION

The step-instruction-over-subroutine command causes DELTA to execute one instruction, display the address of the next instruction, and display that next instruction and its operands.

This command also sets a flag that causes subsequent close-and-display-next or open-and-display-indirect-location commands to perform instruction decoding. The open-and-display command clears the flag.

If the next instruction is BSBB, BSBW, JSB, CALLG, or CALLS, DELTA executes the subroutine and displays the instruction to which the subroutine returns control. This command is said to "step over" subroutines.

;M (Set All Processes Writeable)

Sets the address spaces of all processes to be writeable or read-only by your DELTA process. This command can be used only with DELTA. This command requires CMKRNL privilege.

FORMAT

n ;M

argument

n

If 0, your DELTA process can only read locations in other processes; if 1, your process can read or write any location in any process; if not specified, DELTA returns the current value of the M (modify) flag (0 or 1).

DESCRIPTION

This command is useful for changing values in the running system. This is, of course, an activity that must be pursued only with the greatest of care during timesharing. For this reason, your process must have change-mode-to-kernel (CMKRNL) privilege to use this command. It is safest to use this command only on a stand-alone system.

'(Deposit ASCII String)

Deposits the ASCII string at the current address.

FORMAT

'string'

arguments

string
The string of characters to be deposited

DESCRIPTION

The deposit-ASCII-string command deposits **string** at the current location (.) in ASCII format. The second apostrophe is required syntax. It ends the string. All characters typed between the first and second apostrophes, including linefeed and carriage return, are entered as text.

This command stores the characters in eight-bit bytes, and increments the current address (.) by one for each character stored.

This command does not change the prevailing display mode.

Index

N

P

Q

R

S

V

W

X

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line